Python (M1) – Des décorateurs et des itérateurs

TD #4

Rédacteur: Stéphane Vialette

1 Décorateurs

1.1 Mémoization

En informatique, la **mémoization** est une technique d'optimisation de code consistant à réduire le temps d'exécution d'une fonction en mémorisant ses résultats d'une fois sur l'autre. Bien que liée à la notion de cache, la mémoization désigne une technique bien distincte de celles mises enœuvre dans les algorithmes de gestion de la mémoire cache.

Le terme "mémoization" a été introduit par Donald Michie en 1968 et est dérivé du mot latin *memorandum* signifiant *qui doit être rappelé*. Il y a donc derrière le terme mémoization l'idée de résultats de calculs dont il faut se souvenir. Bien que mémoization évoque *mémorisation*, le terme mémoization a une signification particulière en informatique. Une fonction mémoizée stocke les résultats de ses appels précédents dans une table et, lorsqu'elle est appelée à nouveau avec les mêmes paramètres, renvoie la valeur stockée au lieu de la recalculer. Une fonction peut être mémoizée seulement si elle est référentiellement transparente, c'est-à-dire si sa valeur de retour ne dépend que de la valeur de ses arguments. À la différence d'une fonction tabulée, où la table est statique, une fonction mémoizée repose sur une table dynamique remplie à la volée. La mémoization est donc une façon de diminuer le temps de calcul d'une fonction, au prix d'une occupation mémoire plus importante. La mémoization modifie donc la complexité d'une fonction, en temps comme en espace.

Nous allons mettre en œuvre une technique de memoization par décorateurs. Écrire un décorateur `memoized` (unique fonction d'un module `memoized`) permettant de mémoizer la fonction sur laquelle il est appliqué.

Voici un programme de test illustrant l'utilisation de de décorateur (la sortie de ce program me de test suit). Il est téléchargeable sur la page du cours.

```
import time
from memoized import memoized

N, M = 70, 35

def factorial_without_memoization(n):
    if n in (0, 1):
```

```
        return n
    else:
        return n * factorial_without_memoization(n-1)

for i in range(N, N+2):
    t_s = time.clock()
    res = factorial_without_memoization(i)
    t_e = time.clock()
    print 'factorial_without_memoization(%d): %f' % (i, t_e - t_s)
    t_s = time.clock()
    res = factorial_without_memoization(i)
    t_e = time.clock()
    print 'factorial_without_memoization(%d): %f' % (i, t_e - t_s)

def fibonacci_without_memoization(n):
    if n in (0, 1):
        return n
    else:
        return fibonacci_without_memoization(n-1) + \
            fibonacci_without_memoization(n-2)

for i in range(M, M+2):
    t_s = time.clock()
    res = fibonacci_without_memoization(i)
    t_e = time.clock()
    print 'fibonacci_without_memoization(%d): %f' % (i, t_e - t_s)
    t_s = time.clock()
    res = fibonacci_without_memoization(i)
    t_e = time.clock()
    print 'fibonacci_without_memoization(%d): %f' % (i, t_e - t_s)

@memoized
def factorial_with_memoization(n):
    if n in (0, 1):
        return n
    else:
        return n * factorial_with_memoization(n-1)

for i in range(N, N+2):
    t_s = time.clock()
    factorial_with_memoization(i)
    t_e = time.clock()
    print 'factorial_with_memoization(%d): %f' % (i, t_e - t_s)
    t_s = time.clock()
```

```
    factorial_with_memoization(i)
    t_e = time.clock()
    print 'factorial_with_memoization(%d): %f' % (i, t_e - t_s)

@memoized
def fibonacci_with_memoization(n):
    if n in (0, 1):
        return n
    else:
        return fibonacci_with_memoization(n-1) + \
            fibonacci_with_memoization(n-2)

for i in range(M, M+2):
    t_s = time.clock()
    fibonacci_with_memoization(i)
    t_e = time.clock()
    print 'fibonacci_with_memoization(%d): %f' % (i, t_e - t_s)
    t_s = time.clock()
    fibonacci_with_memoization(i)
    t_e = time.clock()
    print 'fibonacci_with_memoization(%d): %f' % (i, t_e - t_s)
```

Voici une sortie possible de ce programme :

```
barbapapa:> time python test_memoized.py
factorial_without_memoization(70): 0.000093
factorial_without_memoization(70): 0.000035
factorial_without_memoization(71): 0.000039
factorial_without_memoization(71): 0.000039
fibonacci_without_memoization(35): 6.045205
fibonacci_without_memoization(35): 6.017493
fibonacci_without_memoization(36): 9.773134
fibonacci_without_memoization(36): 9.861203
factorial_with_memoization(70): 0.000640
factorial_with_memoization(70): 0.000004
factorial_with_memoization(71): 0.000011
factorial_with_memoization(71): 0.000003
fibonacci_with_memoization(35): 0.000248
fibonacci_with_memoization(35): 0.000002
fibonacci_with_memoization(36): 0.000012
fibonacci_with_memoization(36): 0.000003

real        0m31.752s
user        0m31.687s
```

```
sys         0m0.039s
barbapapa:>
```

Bien sûr, les temps mesurés varient d'une exécution à une autre, mais ceux-ci restent dans le même ordre de grandeur. Vous remarquerez en particulier que :

- Les temps d'exécution des deux fonctions `factorial_without_memoization` et `fibonacci_without_memoization` sont sensiblement les mêmes pour un même paramètre.
- Le temps d'exécution du premier `factorial_with_memoization(70)` est plus long que celui du premier `factorial_without_memoization(70)`.
- La memoization est pleinement effective lors de la seconde exécution de la fonction `factorial_without_memoization(70)`.
- Le temps d'exécution du premier `factorial_with_memoization(71)` est plus court que celui du premier `factorial_without_memoization(71)`.
- Le temps d'exécution du premier `factorial_without_memoization(70)` est plus court que celui du second `fibonacci_without_memoization(70)`.

Pourquoi ?

1.2 Mini mini html

Nous nous proposons ici d'utiliser des décorateurs pour agrémenter des chaînes de caractères. Il s'agira simplement de pouvoir ajouter des balises `<b> ... </b>`, `<i> ... </i>`, et `<u> ... </u>`. Ces trois balises seront utilisables via les décorateurs `html_bold`, `html_italics`, et `html_underline`. Un exemple d'utilisation :

```
In [1]: import htmltags

In [2]: @htmltags.html_bold
   ...: def f1(): return 'f1'

In [3]: f1()
Out[3]: '<b>f1</b>'

In [4]: @htmltags.html_italics
   ...: @htmltags.html_bold
   ...: def f2(): return 'f2'

In [5]: f2()
Out[5]: '<i><b>f2</b></i>'

In [6]: @htmltags.html_underline
   ...: @htmltags.html_italics
   ...: def f3(): return 'f3'
```

```
In [7]: f3()
Out[7]: '<u><i>f3</i></u>'

In [8]:
```

Écrire et tester les décorateurs `html_bold`, `html_italics`, et `html_underline`.

Il apparaît rapidement qu'il va être fastidieux d'écrire un décorateur dédié pour chaque type de balise. Nous nous proposons dans un premier temps d'écrire le décorateur `html_tag` qui prend en argument le caractère utilisé dans la balise créée. Un exemple d'utilisation :

```
In [1]: from htmltags import html_tag

In [2]: @html_tag('b')
   ...: def f(): return 'f'

In [3]: f()
Out[3]: '<b>f<b>'

In [4]: @html_tag('b')
   ...: @html_tag('i')
   ...: @html_tag('u')
   ...: def g(): return 'g'

In [5]: g()
Out[5]: '<b><i><u>g<u><i><b>'

In [6]:
```

Écrire et tester le décorateur `html_tag`.

Ce décorateur `html_tag` n'est cependant pas sans défaut. Il est en effet possible d'utiliser n'importe quelle balise, même si celle-ci n'existe pas. Proposer une solution à ce problème (nous pourrons ici supposer que nous nous limitons aux balises `b`, `u`, `i`, `tt`, `code`, `sub`, et `sup`).

1.3 Vérification de type

Écrire le décorateur `typassert` qui permet de vérifier le type de chacun des arguments d'une fonction. Ce décorateur prend donc autant d'arguments que la fonction qu'il décore (il doit lever une exception dans le cas contraire). Pour plus de souplesse, chaque argument du décorateur est un type ou un tuple de types valides. Donc, si le i-ème argument de `typassert` est un type `t`, il doit s'assurer que le i-ème argument de la fonction qu'il décore est également de type `t`, et si le i-ème argument de `typassert` est un tuple de types ($t_1, t_2, \ldots, t_n$w) il doit s'assurer que le i-ème argument de la fonction qu'il décore est également de l'un au moins des types $t_1, t_2, \ldots, t_n$.

Un exemple simple d'utilisation :

```
import asserttype

@asserttype.asserttype(int)
def f_int(int_val):
    pass

@asserttype.asserttype(float)
def f_float(float_val):
    pass

@asserttype.asserttype(str, (int, long, float))
def f_str_number(str_val, number_val):
    pass

class A(object):
    pass

@asserttype.asserttype(A, A)
def f_A_A(a_A_instance, another_A_instance):
    pass

f_int(1)        # ok
#f_int(1.0)     #  exception (float instead of int)
#f_int(1L)      #  exception (long instead of int)
#f_int('abcd')  #  exception (str instead of int)
#f_int(A())     #  exception (str instead of int)

f_float(1.0)      # ok
#f_float(1)       #  exception (int instead of float)
#f_float(1L)      #  exception (long instead of int)
#f_float('abcd')  #  exception (str instead of int)
#f_float(A())     #  exception (str instead of int)

f_str_number('abcd', 1)        # ok
f_str_number('abcd', 1.0)      # ok
f_str_number('abcd', 1L)       # ok
#f_str_number('abcd', 'efgh') #  exception (str instead of number)
#f_str_number(1, 'abcd')      #  exception (int instead of str)

f_A_A(A(), A()) # ok
#f_A_A(1, A())  #  exception (int instead of A)
#f_A_A(A(), 1)  #  exception (int instead of A)
class B(A):
```

```
    pass
f_A_A(B(), B()) # ok
```

2 itertools

group_by_pairs

Écrire le générateur `group_by_pairs` qui permet de retourner les éléments d'un étirable groupés deux par deux avec chevauchements. Un exemple d'utilisation :

```
In [1]: import mygenerators as g

In [2]: g.group_by_pairs
Out[2]: <function mygenerators.group_by_pairs>

In [3]: g.group_by_pairs(xrange(5))
Out[3]: <itertools.izip at 0x10148aab8>

In [4]: list(g.group_by_pairs(xrange(5)))
Out[4]: [(0, 1), (1, 2), (2, 3), (3, 4)]

In [5]:
```

biz_generator

Écrire le générateur `biz_generator` qui prend en argument un itérable et retourne les éléments qui satisfont aux conditions suivantes :

- `biz_generator` ne retourne que des entiers entre 1 et 10,
- `biz_generator` ne retourne un entier que si l'itérable en entrée délivre au moins 3 éléments,
- `biz_generator` ne retourne pas le premier élément délivré par l'étirable en entrée,
- `biz_generator` ne retourne un multiple de 3 qu'à la condition que celui-ci soit précédé d'un nombre pair, et
- `biz_generator` ne retourne un multiple de 4 qu'à la condition que celui-ci soit suivi d'un nombre impair.