Exercices en Python : Programmation fonctionnelle

1 Filtrer des URLs

Écrire une fonction qui filtre une liste d'URLs en ne gardant que les accessibles

1. en utilisant `filter`
2. avec une *list comprehension*
3. avec une *generator expression*

Exemple :

```
urls = [
    'http://www.he-arc.ch',
    'http://www.google.com',
    'www.he-arc',
    'http://www.cff.ch',
    'http://inexistant.ch',
]

for u in checkurls(urls):
    print u
```

doit donner

```
http://www.he-arc.ch
http://www.google.com
http://www.cff.ch
```

Observez-vous une différence de comportement entre vos trois versions ? si oui, laquelle ?

2 For/map+filter/list comprehensions : comparaison de vitesse

En partant d'une liste de *n* éléments (`l=range(n)`), on veut produire une nouvelle liste qui contient

1. `e+1` pour tous les éléments e de la liste de départ (`[0,1,2, ...]`→`[1,2,3, ...]`)
2. seulement les éléments pairs de la liste de départ (`[0,1,2, ...]`→`[0,2,4, ...]`)
3. `e+1` pour tous les éléments pairs e de la liste de départ (`[0,1,2, ...]`→`[1,3,5, ...]`)

Pour chacun de ces cas, écrire

- Une version avec une boucle `for`
- Une version avec `map` et/ou `filter`
- Une version avec une *list comprehension*

Comparer les vitesses d'exécution de ces trois variantes dans chacun des trois cas, pour différentes longueurs de listes.

3 Debug

Écrire un décorateur `debug` qui affiche les entrées et les sorties de la fonction décorée.
Exemple :

```
@debug
def test(x,y,z=0):
    print "x,y,z = %d, %d, %d" % (x,y,z)
    return x+y/z

test(1,2,3)
try:
    test(1,2)
except:
    print "Exception catched!"
test(1, z=3, y=2)
try:
    test()
except:
    print "Exception catched!"
```

doit donner

```
*** Calling test
*** args : (1, 2, 3)
*** kwargs : {}
x,y,z = 1, 2, 3
*** Exiting test normally
*** result : 1
*** Calling test
*** args : (1, 2)
*** kwargs : {}
x,y,z = 1, 2, 0
*** An error occurred in test
*** integer division or modulo by zero
Exception catched!
*** Calling test
*** args : (1,)
*** kwargs : {'y' : 2, 'z' : 3}
x,y,z = 1, 2, 3
*** Exiting test normally
*** result : 1
*** Calling test
*** args : ()
*** kwargs : {}
*** An error occurred in test
*** test() takes at least 2 arguments (0 given)
Exception catched!
```

4 Memoization

Écrire un décorateur qui "mémoize" la fonction décorée, c'est-à-dire

1. Lors d'un appel, la fonction doit stocker en mémoire une correspondance entrées-sortie
2. Lors d'un rappel de la même fonction avec les mêmes paramètres, la fonction va aller récupérer la valeur déjà calculée plutôt que de la recalculer.

Pour simplifier, on ne s'occupera que des arguments positionnels, sans s'occuper des arguments par mot-clé.

Exemple :

```
from time import sleep, time
from contextlib import contextmanager

@contextmanager
def timing():
    start = time()
    yield
    print "*** Time: %.1f" % (time()-start)

@memoize
def test(x,y):
    sleep(2)
    return x+y

with timing():
    print test(1,2)

with timing():
    print test(1,2)

with timing():
    print test(1,3)
```

doit donner

```
3
*** Time : 2.0
3
*** Time : 0.0
4
*** Time : 2.0
```

Note Un tel décorateur n'a de sens que pour une fonction qui, pour des entrées données, donnera toujours la même sortie !

5 Compter les appels

Écrire un décorateur qui décompte le nombre d'appels faits à la fonction décorée :

```
@count_calls
def test():
    print "test called"
```

```
print test.nbcalls
test()
test()
print test.nbcalls
test()
print test.nbcalls
```

doit donner

```
0
test called
test called
2
test called
3
```

6 Caster les arguments

Écrire un décorateur qui, lors de l'appel de la fonction décorée, force les arguments à la signature donnée. Pour simplifier, on ne s'occupera que des arguments positionnels, sans s'occuper des arguments par mot-clé.

Exemple :

```
@cast_args(str,int)
def f(s,i):
    print "String: "+s
    print i+1

f('12',2)
f(12,'2')
```

doit donner

```
String : 12
3
String : 12
3
```

7 Ré-implémenter contextlib.contextmanager

Nous avons vu au cours que le décorateur `contextlib.contextmanager` permet de transformer un générateur en gestionnaire de contexte. Proposez votre propre implémentation d'un décorateur ayant le même comportement.

Remarque Il est très facile de consulter l'implémentation originale de `contextlib.contextmanager`. Évidemment, tout l'intérêt de l'exercice réside dans le fait de le ré-implémenter vous-même, sans aller regarder la solution !