

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://inexistent.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 Compter les occurrences

Écrire une fonction qui compte le nombre d'occurrences d'un caractère donné dans une liste de chaînes et retourne une liste des résultats

1. avec `map` et `lambda`
2. avec une *list comprehension*

Exemple

```
lines = file('test.txt').readlines()  
  
commas = getcharcounts(',', lines)  
  
for count, line in zip(commas, lines):  
    print "(%d) %s" % (count, line),
```

doit donner

```
(4) a,b,c,d,e
(2) 1,2,3
(5) , , , , ,
(2) alkjfdasf, salkfj sad,
```

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

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 caught!
*** 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 caught!

```

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 fait 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
```