

Exercices en Python : Structures avancées

1 Fractions

Écrire un module `fractions` se comportant de la manière suivante :

```
>>> from fractions import Fraction
>>> f = Fraction(2,3)
>>> f
2/3
>>> f = Fraction(denominator=6, numerator=4)
>>> f
2/3
>>> -f
-2/3
>>> g = Fraction(4)
>>> g
4
>>> print f+g
14/3
>>> print f-g
-10/3
>>> print f*g
8/3
```

↔ Pour simplifier une fraction $\frac{a}{b}$, diviser a et b par le plus grand diviseur commun de a et b . Le PGDC est facilement obtenu par l'algorithme d'Euclide :

```
def pgdc(a,b):
    while b:
        a,b = b, a % b
    return a
```

2 Fibonacci

La suite de Fibonacci est la suite infinie de nombres définie par

$$f_n = \begin{cases} 1 & \text{si } n = 0, 1 \\ f_{n-1} + f_{n-2} & \text{sinon} \end{cases}$$

Donc $f = \{1, 1, 2, 3, 5, 8, 13, 21, \dots\}$

1. Écrire un itérateur qui génère la suite de Fibonacci :

```
>>> for f in fibo() :
...     if f > 30 : break
...     print f
1
1
2
3
5
8
13
21
>>>
```

2. Écrire la même chose, mais sous forme de générateur.

3 Parcours d'arbre

Écrire un générateur `inorder` qui permette de parcourir un arbre binaire en ordre infixe, comme dans l'exemple suivant :

```
class Tree:
    def __init__(self, name, left=None, right=None):
        self.name = name
        self.left = left
        self.right = right

t = Tree('0',
        Tree('1',
            Tree('11'),
            Tree('1r')
        ),
        Tree('r',
            Tree('rl',
                Tree('rll'),
                Tree('rlr')
            ),
            Tree('rr',
                Tree('rrl'),
                Tree('rrr')
            )
        )
    )
```

```

)
)
for i in inorder(t):
    print "Next node in in-order:",
    print i

```

qui doit donner en sortie :

```

Next node in in-order : ll
Next node in in-order : l
Next node in in-order : lr
Next node in in-order : 0
Next node in in-order : rll
Next node in in-order : rl
Next node in in-order : rlr
Next node in in-order : r
Next node in in-order : rrl
Next node in in-order : rr
Next node in in-order : rrr

```

4 Quizz

Écrire un générateur `quizz` qui fasse fonctionner le code suivant :

```

capitales = {
    'Suisse': 'Berne',
    'Allemagne': 'Berlin',
    'Italie': 'Rome',
    'France': 'Paris'
}

q = quizz(capitales)
try:
    c = q.next()
    while True:
        answer = raw_input("Capitale de: " + c + "? ")
        c = q.send(answer)
except StopIteration:
    pass

```

Avec en sortie (par exemple) :

```
C:\somedir>python quizz.py
Capitale de:  Allemagne? Berlin
YES :-)
Capitale de:  Italie? Berne
NO :-(
Capitale de:  Suisse? Berne
YES :-)
Capitale de:  Italie? Rome
YES :-)
Capitale de:  France? Paresse
NO :-(
Capitale de:  France? Paris
YES :-)
C:\somedir>
```

Votre générateur devra donc :

- Prendre un dictionnaire en entrée, qui contient des couples question-réponse
- Générer aléatoirement des questions
- Recevoir les réponses proposées
- Reposer les questions dont la réponse est incorrecte
- S'arrêter lorsque chaque question aura reçu exactement une fois sa réponse correcte.

5 Error Logging

On veut écrire un gestionnaire de contexte qui récupère les erreurs et écrit quelques détails à la console. Par exemple :

```
if __name__ == '__main__':

    print "Let's go!"

    with error_catcher():
        for i in range(3,-1,-1):
            print 1.0/i

    print "The End!"
```

doit donner :

```
Let's go!
0.3333333333333333
0.5
1.0
*** An error occurred in line 26!
***     type : <type 'exceptions.ZeroDivisionError'>
***     error : float division
The End!
```

1. Écrire une version sans l'aide de contextlib

Indication La ligne de l'erreur est stockée dans l'attribut `tb_lineno` des objets de type `traceback`.

2. Écrire une version avec l'aide du décorateur `@contextmanager` de contextlib

Indication Le triplet (`type`, `value`, `traceback`) peut être récupéré par la fonction `sys.exc_info()`

6 Génération d'XML

Écrire un gestionnaire de contexte (selon la méthode de votre choix) qui ouvre et referme automatiquement des tags XML (sur la sortie standard) :

```
if __name__ == '__main__':  
  
    with tag('L1', attr1=1, attr2=2):  
        print "in L1"  
        with tag('L2'):  
            print "in L2"  
            print "in L1 again"  
        print "out"
```

qui donne :

```
<L1 attr2=2 attr1=1>  
in L1  
<L2>  
in L2  
</L2>  
in L1 again  
</L1>  
out
```

7 Conteneur générique

Écrire une classe `GenericContainer` dont les attributs des instances sont déterminés dynamiquement lors de l'appel du constructeur, en utilisant des arguments nommés :

```
>>> c = GenericContainer(toto=1, test="string")
>>> c.toto
1
>>> print c.test
string
>>> c.blah
Traceback (most recent call last) :
File "<stdin>", line 1, in?
AttributeError : GenericContainer instance has no attribute
'blah'
>>> c2 = GenericContainer(blah=1)
>>> print c2.blah
1
>>> c2.toto
Traceback (most recent call last) :
File "<stdin>", line 1, in?
AttributeError : GenericContainer instance has no attribute
'toto'
```

↪ Voir le *built-in* `setattr(object, name, value)`.