Tutoriel : Python unit test

Introduction

Durant ce tutoriel, nous vous présenterons comment effectuer une série de tests unitaires en langage Python. Le framework utilisé pour ces tests est pyunit.

Il est intégré directement à partir de la version 2.1 de python contrairement à la version 2 où il vous sera nécessaire de la télécharger.

Pour vous expliquer le fonctionnement des tests unitaire sur python, nous avons choisis un exercice de conversion de nombre en chiffres romains et inversement.

Nous vous fournirons les fichiers sources roman.py et son fichier de tests romantest.py.

Ils vous faudra réparer des erreurs glissées à l'intérieur des tests.

Tutoriel

1) Le fichier roman .py

Tout d'abord, on importe la librairie des expressions régulières qui nous servirons par la suite.

```
import re
```

On définit les différentes exceptions à lever.

```
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass
```

On créé un tableau contenant les chiffres romains et leurs valeurs transcrites en chiffres arabes.
Les chiffres romains ne prenaient pas en compte ceci :

- ➔ Le zéro
- ➔ Les chiffres négatifs
- ➔ Les chiffres supérieurs à 3999 car il est interdit d'écrire quatre fois le même symbole d'affilée.

```
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))
```

On peut maintenant créer la méthode de conversion d'un nombre en chiffres romains qui vérifiera par une gestion d'exception que le nombre envoyé en paramètre est bien compris entre 1 et 4999 et aussi qu'il ne soit pas un nombre décimal.

La boucle for, elle, s'occupera de la conversion en incrémentant le résultat avec les valeurs en chiffres romains.

```
def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 5000):
        raise OutOfRangeError("number out of range (must be 1..4999)")
    if int(n) != n:
        raise NotIntegerError("decimals can not be converted")

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

On utilise, maintenant, la librairie des expressions régulières que nous avons importée précédemment pour créer un pattern permettant de vérifier la validité des chiffres romains. On ne pourra par exemple, pas avoir plus de 3 fois la même lettre.

```
#Define pattern to detect valid Roman numerals
romanNumeralPattern = re.compile("""
    ^                   # beginning of string
    M{0,3}              # thousands - 0 to 3 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
                        #            or 500-800 (D, followed by 0 to 3 C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                        #        or 5-8 (V, followed by 0 to 3 I's)
    $                   # end of string
    """ ,re.VERBOSE)
```

Voici pour finir la fonction inverse de la première, à savoir, une fonction permettant de transformer des chiffres romains en nombres arabes. On vérifiera premièrement

sous peine de lever une exception, la validité de ce qui nous a été envoyé en paramètre et deuxièmement la conversion en elle même.

```
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError('Input can not be blank')
    if not romanNumeralPattern.search(s):
        raise InvalidRomanNumeralError('Invalid Roman numeral: %s' % s)

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

2) Le fichier romanTest.py

Nous allons écrire une suite de tests qui évalue ces fonctions et s'assure qu'elle se comporte comme l'on voudrait qu'elles le fassent.

On importe tout d'abord le fichier roman.py ainsi que la librairie de test "unittest".

```
import roman
import unittest
```

Il faut ensuite préparer des valeurs à tester afin de les utiliser dans les fonctions de test de la classe knownvalues dans laquelle nous enverrons en paramètre " unittest.TestCase " pour bien préciser que celle-ci est une classe de test.

```
class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D')
```

On utilisera ensuite dans les différentes classes de test des méthodes qui s'occuperont de tester les différentes erreurs possibles dans le fichier roman.py

En python, le premier paramètre utilisé par une méthode doit toujours être une référence d'instance. Vous pourriez en principe utiliser un nom de variable quelconque pour ce paramètre, mais il est vivement conseillé de respecter la convention qui consiste à toujours lui donner le nom **self**. Le paramètre **self** désigne donc l'instance à laquelle la méthode sera associée, dans les instructions faisant partie de la définition. (De ce fait, la définition d'une méthode comporte toujours au moins un paramètre, alors que la définition d'une fonction peut n'en comporter aucun).

Dernière chose importante, il ne faudra surtout pas supprimer ces lignes

```
if __name__ == "__main__":
    unittest.main()
```

Elles permettent de lancer et reconnaître les méthodes tests dans la console de python idle.

Nous avons glissé des erreurs dans les tests fournis. A vous de les retrouver et de remettre en place des tests fonctionnels.

Nous précisons bien que vous n'aurez pas à réécrire les fonctions de tests mais seulement a détecter des petites erreurs glissées à l'intérieur.

Cela vous permettra de bien comprendre l'utilité de ces tests.

Vous devrez obtenir ce résultat lors du lancement de ceux-ci :

```
............
----------------------------------------
Ran 12 tests in 0.326s

OK
>>>
```

3) Sources

- Dive into python
- Classes méthodes héritage en python