Dépôt BLIH: PYjour03
Répertoire de rendu : racine du dépôt.

dumbXML : le format qui vous veut du bien :P

Dernière étape importante lorsqu'on étudie les processus de traitement de texte. Les principes de sérialisation/dé-sérialisation. Nous allons prendre l'état mémoire d'un AST et le sauvegarder dans un format texte consultable (pour débogage, compréhension) dans un fichier. Nous sérialisons notre AST. De même, nous pouvons souhaiter charger un fichier et le projeter en mémoire. C'est la dé-sérialisation.
Il existe en Python des formats et des modules qui remplissent cet objectif (pickle, json, bson, …), toutefois c'est l'occasion pour aborder un dernier format de fichier : le format XML.

Il faut fournir un module nommé **dumbXml.py** .

to_dxml :

- Le module va grâce au décorateur 'add_method' du module **pyrser.meta** ajouter une méthode **to_dxml** à la classe **pyrser.node.Node**.
  Pour information, cette technique est de l'extension de classe dynamique (**monkey patching**... internet est ton ami) et à pour but de rajouter des fonctionnalités à un type sans toucher à sa définition initiale. Ici cette altération ne sera effective que lorsque l'utilisateur chargera (import) **pyrser.node.Node** ET **dumbXml**.
  Pour voir un exemple d'utilisation de @meta.add_method consulter les sources de pyrser :

  http://pythonhosted.org/pyrser/_modules/pyrser/passes/dumpParseTree.html

  La méthode **to_dxml** ainsi attaché à la classe **Node** permet de dumper les arbres construits avec pyrser ayant pour racine un Node.

```
import dumbXml
from pyrser.parsing import node

x = node.Node()
x.txt = "cool"
x.flags = True
x.subnode = node.Node()
x.subnode.num = 12
x.zero = None
x.sb2 = node.Node()
x.sb2.real = 3.4e+20
print(«RES:\n%s» % x.to_dxml())
```

fournira le résultat suivant :

```
iopi$ python3.3 test_to_dxml.py
RES:
<.root type = object>
    <flags bool = True/>
    <sb2 type = object>
        <real float = 3.4e+20/>
    </sb2>
    <subnode type = object>
        <num int = 12/>
    </subnode>
    <txt str = 'cool'/>
    <zero/>
</.root>
iopi$
```

Cet exemple présente des types scalaires à gérer. Gérons aussi les types collections.

```
import dumbXml
from pyrser.parsing import node

tree = node.Node()
tree.ls = [1, 2.0, "titi", True, [2, 3, 4, [3, [3, 4]], 5]]
tree.dct = {"g":1, "y":2, "koko":{'D', 'T', 'C'}}
tree.aset = {'Z', 'X', 'T', 'U'}
tree.ablob = b'\xFF\xaa\x06Th -}'
print(tree.to_dxml())
```

Qui produira le

```
<.root type = object>
    <ablob type = blob>
        FF AA 06 54 68 20 2D 7D
    </ablob>
    <aset type = set>
        <'T'/>
        <'U'/>
        <'X'/>
        <'Z'/>
    </aset>
    <dct type = dict>
        <.idx __key = 'g' int = 1/>
        <.idx __key = 'koko' type = set>
            <'C'/>
            <'D'/>
            <'T'/>
        </.idx>
        <.idx __key = 'y' int = 2/>
    </dct>
    <ls type = list>
        <.idx __value = 0 int = 1/>
        <.idx __value = 1 float = 2.0/>
        <.idx __value = 2 str = 'titi'/>
        <.idx __value = 3 bool = True/>
        <.idx __value = 4 type = list>
            <.idx __value = 0 int = 2/>
            <.idx __value = 1 int = 3/>
            <.idx __value = 2 int = 4/>
            <.idx __value = 3 type = list>
                <.idx __value = 0 int = 3/>
                <.idx __value = 1 type = list>
                    <.idx __value = 0 int = 3/>
                    <.idx __value = 1 int = 4/>
                </.idx>
            </.idx>
            <.idx __value = 4 int = 5/>
        </.idx>
    </ls>
</.root>
```

dxmlParser :

- Ce qui est fait dans un sens peut être fait dans l'autre.

```
import dumbXml

## READ SAMPLE OF ALL SUPPORTED TYPE

dxml = dumbXml.dxmlParser()

xml0 = dxml.parse("""
        <.root type=object>
            <raw str='From scratch' />
        </.root>
    """)

if xml0.raw == "From scratch":
    print("OK")

xml1 = dxml.parse_file("t1.dxml")

if xml1.subnode.num == 12:
    print("OK")

xml2 = dxml.parse_file("t2.dxml")

if xml2.ls[4][3][1][1] == 4:
    print("OK")
```

Remarques :

- Le format DXML prends certaine liberté avec le vrai XML. Le principe générale est là.
- Les attributs, les clefs des dictionnaires (et set) sont triées avant d'être généré.
- Les clefs (dict et set) sont forcément des strings.
- Les fichiers de référence sont sur l'intra ainsi que les codes sources d'exemple.
- N'ouvrez pas les fichiers par le navigateur (vous verrez rien). Téléchargez-les !
- Ces exemples présentes tous les types à surcharger. Par contre pas toutes les encapsulations.
- Penser à des fonctions récursives pour le dump.
- Penser à un règle récursive pour le parsing (la partie principal pas les détails).
- Dans un hook le **self** est l'instance du parseur en cours avec le bon stream etc... Il se peut que vous souhaitiez consommer du contenu DANS UN HOOK. Vous pouvez directement appeler certaines fonctions de BasicParser: http://pythonhosted.org/pyrser/base.html#pyrser.parsing.parserBase.BasicParser
  Il se trouve que pour associer le tag fermant du tag ouvrant c'est particulièrement adapter. D'autres utilisations abusives sont à proscrire.
- Vous n'êtes pas obligés de gérer l'indentation. La mouli de correction n'en tiendra pas compte.
- ESSAYEZ DE FAIRE LE DUMPEUR ET LE PARSEUR EN PARALLÈLE :
  - Vous voyez les problèmes progressivement.
  - Vous avez plus de point par la moulinette sans bloquer.