Python et XML

XML, avec une vision DOM

Langages de balises

- Au commencement était SGML
  - *Standard Generalized Markup Language*
  - Simplifié énormément avec HTML pour le Web
  - Transformé en produit marketing avec XML
- Python supporte toute la petite famille
  - HTML, XHTML et SGML pour HTML
  - XML, DOM et SAX

XML, quels outils?

Il existe plusieurs librairies pour manipuler des documents XML avec Python, mais ce chapitre se limite à une partie du package standard `xml`. Ce package fournit plusieurs sous-packages:

- `dom` contient `minidom`, une implémentation de DOM en Python,
- `sax` est un dictionnaire des implémentation disponibles,
- `parsers` contient les parsers utilisés en interne par DOM et SAX.

En compléments, plusieurs librairies et extensions sont disponibles. Elles fournissent, par exemple, des abstractions de plus haut niveau et des outils typés par rapport à une utilisation donnée.

DOM, petit rappel

Pour ce qui est de la manipulation d'un fichier XML, nous allons uniquement utiliser DOM (Document Object Model). DOM propose de manipuler un document XML en mémoire comme un arbre d'objets représentant les noeuds du document. Les interfaces standard DOM sont les suivantes.

Interface	**Représentation de**
Node	Interface de base des noeuds
NodeList	Séquence de noeuds
Document	Représente un document complet
Element	Elément de la hiérarchie
Attr	Noeud représentant un attribut d'un noeud
Comment	Noeud repréntant un commentaire
Text	Noeud de données textuelles

Exemple du chapitre

Le listing suivant présente un exemple de document XML utilisé dans la suite du chapitre pour illustrer la manipulation de tels documents.

```
<?xml version="1.0" ?>
<contacts>
  <contact name="doe" firstname="john">
    <address>
      <road value="10, binary street" />
      <postal value="0001" />
      <city value="cpu" />
    </address>
    <programming lang="asm" />
  </contact>
  <contact name="dupont" firstname="jean">
    <address>
      <road value="impasse de l'assembleur" />
      <postal value="0100" />
      <city value="dram" />
    </address>
    <programming lang="c" />
  </contact>
  <contact name="terprette" firstname="quentin">
    <address>
      <road value="avenue du script" />
      <postal value="1001" />
      <city value="salt snake city" />
    </address>
    <programming lang="python" />
  </contact>
</contacts>
```

Naviguer dans un arbre DOM

`minidom` «il fait le maximum»

Le module `minidom` est une mise en oeuvre de DOM pour Python. Il fournit toutes les interfaces de base DOM et un parser de fichier (ou de chaîne) XML. Il est disponible en standard dans l'environnement (librairie PyXML).

Parser un document XML

La fonction `minidom.parse` permet d'analyser un fichier XML et d'obtenir un arbre DOM correspondant. Elle retourne un objet de type `Document`. En une ligne, elle est pas belle la vie?

```
>>> from xml.dom import minidom
>>> doc = minidom.parse('/tmp/contacts.xml')
>>> doc
<xml.dom.minidom.Document instance at 0x...>
```

Parcourir un arbre DOM

Un certain nombre d'opérations et d'attributs disponibles sur tous les éléments d'un arbre DOM pour en permettre son parcours.

La fonction `hasChildNodes()` indique si un noeud a des fils.

```
>>> doc.hasChildNodes()
True
```

L'attribut `childNodes` donne accès aux fils d'un noeud (sous la forme d'une liste).

```
>>> doc.childNodes
[<DOM Element: contacts at 0x...>]
>>> doc.childNodes[0].hasChildNodes()
True
```

L'attribut `documentElement` fournit le noeud racine d'un document DOM.

```
>>> root = doc.documentElement
```

Les attributs `firstChild, lastChild` donnent accès respectivement au premier et au dernier fils d'un noeud.

```
>>> root.childNodes
[<DOM Text node "u'\n  '">, <DOM Element: contact at 0x...>, <DOM Text node "u'\n  '
>>> root.firstChild
<DOM Text node "u'\n  '">
>>> root.lastChild
<DOM Text node "u'\n'">
```

Les attributs `nextSibling, previousSibling` donnent respectivement accès au fils suivant et au fils précédent (ou `None` si plus de fils). Le calcul se fait par rapport à une racine commune.

```
>>> current = root.firstChild
>>> while current:
...     print current
...     current = current.nextSibling
...
<DOM Text node "u'\n  '">
<DOM Element: contact at 0x...>
<DOM Text node "u'\n  '">
<DOM Element: contact at 0x...>
<DOM Text node "u'\n  '">
<DOM Element: contact at 0x...>
<DOM Text node "u'\n'">
```

L'attributs `parentNode` donne accès au parent (direct) d'un noeud.

```
.. doctest::
```

```
>>> root
<DOM Element: contacts at 0x...>
>>> root.firstChild
<DOM Text node "u'\n  '">
>>> root.firstChild.parentNode
<DOM Element: contacts at 0x...>
```

La fonction `isSameNode()` permet de tester si deux noeuds sont égaux (égalité au sens de l'identité, parle-t-on de la même personne).

```
>>> root.firstChild.isSameNode(root.lastChild)
False
>>> root.firstChild.isSameNode(root.childNodes[0])
True
```

Recherche dans un arbre DOM

La recherche dans un arbre DOM se fait principalement par nom de tag (nom de noeud). La fonction `getElementsByTagName()`, disponible sur les noeuds de type `Document` et `Element`, donne tous les fils (et sous-fils et descendants) portant le nom de tag fourni.

```
>>> for element in root.getElementsByTagName('contact'):
...     print element
<DOM Element: contact at 0x...>
<DOM Element: contact at 0x...>
<DOM Element: contact at 0x...>

>>> for element in root.getElementsByTagName('programming'):
...     print element
<DOM Element: programming at 0x...>
<DOM Element: programming at 0x...>
<DOM Element: programming at 0x...>
```

`NodeList` et objets séquences

Le résultat retourné par les fonctions `childNodes` et `getElementsByTagName` est de type `NodeList` et se comporte comme une séquence «classique».

> Pour se comporter comme une séquence, un objet doit offrir (au minimum) les méthodes suivantes:
>
> - `__len__()` et `__getitem__(i)` pour l'accès aux éléments (boucle `for`),
> - `__setitem__(i)` et `__delitem__(i)` pour les séquences modifiables.

Accéder aux informations d'un noeud

Informations liées au noeud

L'attribute `nodeType` donne une constante (1-10) représentant le type du noeud: `ELEMENT_NODE`, `ATTRIBUTE_NODE`, `TEXT_NODE`, `CDATA_SECTION_NODE`, `ENTITY_NODE`, `PROCESSING_INSTRUCTION_NODE`, `COMMENT_NODE`, `DOCUMENT_NODE`, `DOCUMENT_TYPE_NODE`, `NOTATION_NODE`.

```
>>> root.nodeType == minidom.Node.ELEMENT_NODE
True
>>> root.firstChild.nodeType == minidom.Node.ELEMENT_NODE
False
>>> root.firstChild.nodeType == minidom.Node.TEXT_NODE
True
```

L'attribute `nodeName` donne le nom du n{oe}ud (tag de la balise). Attention, le comportement est variable selon les noeuds: pour un élément ce sera le nom de la balise XML, pour un

noeud représentant du texte, ce sera `#text`.

```
>>> root.nodeName
u'contacts'
>>> root.firstChild.nodeName
'#text'
```

L'attribut `nodeValue` donne la valeur du n{oe}ud (contenu). Attention, le comportement est variable selon les noeuds: pour un élément ce sera `None` pour un noeuds contenant de l'information ce sera cette information. Pour les noeuds contenant du texte, il est aussi possible d'utiliser l'attribut `data`.

```
>>> root.nodeValue
>>> root.firstChild.nodeValue
u'\n  '
>>> root.firstChild.data
u'\n  '
```

Informations liées aux attributs d'un noeud

La fonction `hasAttributes()` teste la présence d'attributs dans un noeud.

```
>>> root.hasAttributes()
False
>>> root.childNodes[1].hasAttributes()
True
```

L'attribute `attributes` fourni les attributs du noeud (objet du type `NamedNodeMap` ou `None`).

```
>>> root.firstChild.attributes
>>> attributes = root.childNodes[1].attributes
>>> attributes
<xml.dom.minidom.NamedNodeMap object at 0x...>
```

La fonction `getAttribute()` retourne la valeur d'un attribut comme une chaîne.

```
>>> c = root.childNodes[1]
>>> c.getAttribute('name')
u'doe'
```

La fonction `getAttributeNode()` retourne la valeur d'un attribut comme un objet de type `Attr`.

```
>>> c.getAttributeNode('name')
<xml.dom.minidom.Attr instance at 0x...>
```

Une `NamedNodeMap` se comporte comme un dictionnaire. La méthode `item()` permet d'accéder aux couples nom d'attribut / valeur textuelle d'attribut, la méthode `keys()` donne accès aux noms des attributs et la méthode `values()` donne accès à la liste des objets `Attr` représentant les attributs.

```
>>> attributes.items()
[(u'name', u'doe'), (u'firstname', u'john')]
>>> attributes.keys()
```

```
[u'name', u'firstname']
>>> attributes.values()
[<xml.dom.minidom.Attr instance at 0x...>, <xml.dom.minidom.Attr instance at 0x...>]
>>> for a in attributes.values():
...     print a.nodeType, a.nodeName, a.nodeValue
...
2 name doe
2 firstname john
```

En plus des méthodes classiques sur les dictionnaires, il existe deux méthodes pour manipuler les `NamedNodeMap`:

- `length` donne la taille du dictionnaire d'attributs,
- `item(i)` donne le i-eme élément (de type `Attr`) du dictionnaire ou `None`.

```
>>> attrs = root.childNodes[1].attributes
>>> for idx in range(0, attrs.length):
...     a = attrs.item(idx)
...     print '(' + a.name + ')',
...     print a.nodeType, a.nodeName, a.nodeValue
...
(name) 2 name doe
(firstname) 2 firstname john
```

Construire un document XML

Créer un arbre DOM

La création d'un document (de son arbre DOM) se fait par instanciation de la classe `Document` (c'est un objet comme un autre). Cette classe fournit des méthodes pour fabriquer les noeuds. A la création, un document ne contient rien, pas même de noeud racine.

```
>>> newdoc = minidom.Document()
>>> newdoc
<xml.dom.minidom.Document instance at 0x...>
>>> newdoc.documentElement
>>>
```

Créer des noeuds DOM

Les méthodes de création de noeud n'incluent pas l'ajout de l'élément créé dans l'arbre. Ici, nous allons créer un document excessivement simple:

- une racine `root` avec un attribut `name`,
- un commentaire,
- un noeud de texte contenu dans un noeud `sometext`.

La fonction `createElement(tag)` crée un nouvel élément (de type `Element`) avec le nom de tag passé en paramètre et le retourne.

```
>>> newroot = newdoc.createElement('root')
```

La fonction `createAttribute(name)` crée un noeud de type `Attr` avec le nom `name`. Une fois le noeud créé, la valeur peut être fixée avec l'attribut `nodeValue`.

```
>>> rootattr = newdoc.createAttribute('name')
>>> rootattr.nodeValue = 'foo'
```

La fonction `createTextNode(data)` crée un noeud de donnée contenant le texte passé en paramètre (à inclure dans un noeud englobant, ici le noeud `sometext` créé pour).

```
>>> textnode = newdoc.createElement('sometext')
>>> text = newdoc.createTextNode('this node\ncontains text')
>>> textnode.appendChild(text)
<DOM Text node "'this node\n'...">
```

La fonction `createComment(text)` crée un noeud de commentaires contenant le texte passé en paramètre.

```
>>> comment = newdoc.createComment('a very usefull comment')
```

Ajout de noeuds dans un arbre

Les méthodes d'ajout dans un arbre viennent en complément des méthodes de création. Leur dissociation est due aux multiples usages que l'on peut faire d'un noeud. Il est possible de construire un document XML partir d'un document existant, dans ce cas on ne va pas recréer tout les noeuds, mais peut être simplement les réorganiser.

La fonction `appendChild(new)` ajoute un élément à la fin de la liste des fils d'un noeud.

```
>>> newdoc.appendChild(newroot)
<DOM Element: root at 0x...>
>>> textnode.appendChild(text)
<DOM Text node "'this node\n'...">
>>> newroot.appendChild(textnode)
<DOM Element: sometext at 0x...>
```

La fonction `insertBefore(new, old)` ajoute un élément avant un fils donné d'un noeud.

```
>>> newroot.insertBefore(comment, textnode)
<DOM Comment node "'a very use'...">
```

La fonction `replaceChild(new, old)` remplace un élément fils d'un noeud par un autre élément.

La fonction `setAttribute(name, value)` crée un nouvel attribut sur un noeud sans passer par une instance de type `Attr`.

```
>>> newroot.setAttribute('usefull', 'nop')
```

La fonction `setAttributeNode(new)` ajoute un noeud attribut au noeud considéré.

```
>>> newroot.setAttributeNode(rootattr)
```

Supprimer des noeuds d'un arbre DOM

Il est aussi possible de supprimer des noeuds dans un arbre DOM, par exemple pour en créer une nouvelle version plus épurée ou pour une restructuration d'un document XML.

La fonction `removeChild(old)` supprime un fils d'un noeud à utiliser conjointement avec `unlink()`.

```
>>> try:
...     old = root.removeChild(root.firstChild)
...     old.unlink()
... except ValueError: print 'failed'
...
```

La fonction `removeAttribute(name)` supprime un attribut d'un noeud en le désignant par son nom.

```
>>> root.firstChild
<DOM Element: contact at 0x...>
>>> root.firstChild.removeAttribute('firstname')
```

La fonction `removeAttributeNode(old)` supprime un attribut d'un noeud par la référence de l'objet `Attr` le représentant ou lève l'exception `NotFoundErr`.

Sérialiser un document XML

L'objectif de faire un document XML est d'obtenir un fichier. Les fonctions `toxml()`, `toprettyxml()` donnent la version texte d'un arbre DOM (une ligne ou multi-ligne indentées). Elles sont disponibles sur tous les noeuds et l'évaluation est automatiquement récursive. Pour sérialiser un document, il suffit d'appeler une de ces méthodes sur l'objet document. Si l'appel est fait sur un noeud, alors le fichier sera un sous ensemble du document XML.

```
>>> newdoc.toxml()
'<?xml version="1.0" ?><root name="foo" usefull="nop"><!--a very usefull comment--><
```

Et la version lisible

```
>>> print newdoc.toprettyxml()
<?xml version="1.0" ?>
<root name="foo" usefull="nop">
    <!--a very usefull comment-->
    <sometext>
            this node
contains text
    </sometext>
</root>
```

Exercices

Le fichier `etudiants.xml` contient 400 dossiers d'étudiants (générés aléatoirement) avec la structure décrite ci-dessous. Le but de l'exercice est de produire trois listes sous forme de

fichiers XML contenant (par ordre décroissant pour les deux premières) les étudiants admissibles, sur liste complémentaire ou refusés.

```
<?xml version="1.0" ?>
<etudiants>
  <etudiant nom="doe" prenom="john" dossier="0">
    <origine universite="mit" />
    <formation discipline="informatique" niveau="4" />
    <resultats moyenne="16" />
  </etudiant>
</etudiants>
```

Pour les trois listes, les informations conservées sont le nom, prénom, l'université d'origine et les résultats. Les règles de sélection sont les suivantes.

- pour être admis, un(e) étudiant(e) doit avoir un niveau Bac+4, dans une formation en informatique, avec une moyenne supérieure à 12;
- si sa discipline n'est pas informatique (mais toujours avec les critères de niveau et de moyenne), il (elle) est sur liste complémentaire;
- sinon il(elle) est refusé(e).

Il faut pour cela parser le fichier `etudiants.xml`, naviguer dans l'arbre DOM ainsi produit et construire trois arbres DOM qui seront en fin de traitement sérialisés dans les fichiers `admissibles.xml`, `complementaires.xml` et `refuses.xml`. La structure de ces trois fichiers est décrite ci-dessous.

```
<?xml version="1.0" ?>
<admissibles|complementaires|refuses>
  <etudiant nom="doe" prenom="john" dossier="0">
    <origine universite="mit" />
    <resultats moyenne="16" />
  </etudiant>
</admissibles|complementaires|refuses>
```