Python/XML HOWTO

The Python/XML Special Interest Group

xml-sig@python.org

도운이

송재경

ssaky@kldp.org

XML 은 SGML 의 서브셋인, 확장가능한 조판언어(eXtensible Markup Language) 로서, 특정한 어플리케이션 조판언어를 생성하고 처리하는것을 가능하게 해준다. Python 은 XML 데이터를 처리하는데 매우 훌륭한 언어이다. 이 문서는 Python/XML 패키지의 튜터리얼로서, 이글을 읽는 독자가 이미 XML 의 구조와 전문용어에 익숙하다고 가정하여 쓰여졌다. 이문서는 초안으로서 'XXX' 라고 표현되는 부분은 후에 채워지거나, 다시 쓰이거나, 수정될 부분이다.

1. XML 의 시작

XML 은 eXtensible Markup Language 로서 SGML(Standardized General Markup Language)의 부분집합이다. XML 은 쉽게 실행하고 사용하는것을 지향하며, 이미 새로운 다양한 표준을 위한 특수한 마크업 언어로 사용되고 있다. 여기에는 수학적인 등식을 표현하기 위한 MathML, 멀티미디어를 표현하기 위한 Synchronized Multimedia Integration Language 등이 있다.

SGML 과 XML 은 문서의 다양한 구성요소들을 그들의 기능과 의미과 함께 태그화하여 표현한다. 예를 들어 레포트는 여러가지 부분으로 나뉠수있다. : 그것은 제목(title) 과 한명이상의 작성자(author), 서문(abstract), 실제 레포트의 내용, 참고자료 목록 등으로 나눌수 있을것이다. 이러한 레포트를 작성하기 위한 마크업 언어는, 그 내용이 서문인지, 제목인지등을 가리킬수있는 태그를 가지고 있어야 한다. 이렇게 함으로서, 어떻게 문서가 실제적으로 출력될지에 대한 물리적인 세부기술에 혼동을 주지 않을것이다. 서문은 문서의 다른 부분보다 작은 폰트와 좁은 폭의 여백으로 출력되어야 할것이다. 그러나 마크업은 보통 이렇게 자세한 세부기술을 하지는 않는다.; 다른 프로그램을 사용하여 마크업 언어를 TEX 와 같은 식자언어로 변환할 수 있을것이고, 정교하게 처리할 수 있을것이다.

XML 은 HTML 과 매우 흡사한 모양을 갖는다; 문서는 하위요소를 포함하는 하나의 요소(element)로 구성된다. 그리고 그 하위요소들은 그들 자신의 내용으로 또 하위요소들을 가질 수 있다. 요소는 문장안에서 < > 안에 존재하는 태그로 표현된다. 요소에는 두가지 표현 방법이 있다. 그중 하나는 <name>Euryale</name> 와 같이 시작태그와 마침태그가 존재하는것

이다. 이것은 "Euryale" 내용(content)을 갖는 name 요소를 나타낸다. 이러한 내용에는 일반적인 텍스트 데이터, XML 요소, 또는 이 두가지가 혼합되어 나타날 수 있다. 요소는 또한 아무런 내용도 가지지 않는 공백일수도 있다. 이러한 때에는 <stop/> 과 같이 슬래쉬로 끝나는 하나의 태그로 표현된다. HTML 과는 달리 XML 은 대소문자를 구분한다. 그러므로 stop 과 Stop 은 서로 다른 두개의 요소로 인식된다.

시작태그와 공백태그는, 그 요소와 함께하는 특정값을 갖는 속성(attribute)을 가질 수 있다. 예를 들어 <name lang='greek'>Herakles</name> 은 그 값이 "greek" 인 lang 속성을 갖는 name 요소를 나타낸다. 이것은 속성값이 'latin' 인 <name lang='latin'>Hercules</name> 과 비교될 수 있을것이다.

XML 언어는 DTD (Document Type Definition) 로 정의된다. DTD 는 요소가 가질 수 있는 이름과, 어떤 요소가 서로 중첩될수 있는가등을 정의해준다. DTD 는 또한 각 요소에 나타날수있는 속성과 그 기본값, 그리고 그 값이 생략할 수 있는 값인지 등에 대해 정의한다. HTML 을 예로 들면, LI 요소는 리스트의 목록을 표현할 때 사용되어지는데, 리스트를 표현하는 OL 또는 UL 요소안에서만 사용해야한다. 적합한 파서는 DTD 와 문서를 넘겨받아, 그 문서가 DTD 의 법칙에 잘 맞는지를 검증해주거나, 하나 또는 그 이상의 법칙에 위배되었다는 것을 결정해주게 된다.

XML 을 처리하기 위한 어플리케이션은 두가지 타입으로 구분되어질 수 있다. 가장 간단한 부류는 특정한 하나의 마크업 언어를 조작하기 위한 어플리케이션이다. 예를 들어, 화학관련 프로그램은 MathML 이 아닌, 화학 마크업 언어만을 처리하기를 원할것이다. 이러한 어플리케이션은 하나의 DTD 로 명세화 할수 있으며, 여러종류의 마크업 언어를 핸들링하기 위한 능력은 필요없을 것이다. 이러한 부류는 작성하기가 쉬우며, 이용가능한 파이썬 소프트웨어로 구현하기가 쉬운편이다.

두번째 부류는 좀 더 범용적이며, 어떠한 마크업 언어에 대해서도 핸들링이 가능한 것이다. 선택된 DTD 를 따르는 XML 을 작성하도록 도와주는 '스마트 XML 에디터' 같은 프로그램을 예로 들어보자. 이것은 허용되지 않는 요소를 입력하는것을 막아주어야 할것이고, 커서위치에 놓일수 있는 적절한 요소들을 제안할 수 있을것이다. 이러한 어플리케이션은 어떠한 가능한 XML 기반의 마크업도 핸들링 가능해야 하며, 그러기 위해서는, 사용되는 DTD 를 포함하는 데이터 구조를 가지고 있어야 할것이다. XXX 이러한 형태의 어플리케이션은 현재로선 파이썬으로 구현하기는 어려운 상황이다. (XXX DTD 모듈이 추가되기를 지켜봐야 할것이다...)

XML 의 문법에 대한 자세한 자료는, http://www.w3.org/TR/xml-spec.html 에서 XML 1.0 명세서를 볼 수 있을것이다. 모든 명세서와 마찬가지로, 이것은 매우 형식적이며, XML 을 시작하거나 배우기 위해 보기에는 적합하지가 않다. 표준에 주석이 붙은 버전인 http://www.xml.com/xml/pub/axml/axmlintro.html 이 더 이해하기 쉬울것이다. 또한 여기에는 XML 을 시작하려는 사람에게 적합한 튜토리얼등의 다양한 문서들도 있다.

이 HOWTO 의 나머지 부분에서는, 이미 적절한 전문용어에 익숙하다고 생각할것이며, 대부분의 섹션에서 요소(element) 와 속성(attribute) 등의 XML 용어들을 사용할 것이다. 문서 객체 모델을 다루는 섹션 4에서는, 당신이 관련 문서들을 읽어봤으며, 노드(Node) 등과 같은 것에 친숙하다고 가정할것이다. 그러나 섹션 3을 위해 특별히 Java SAX 도구에 대한 경험을 요구하지는 않는다.

1.1. 관련 홈페이지

2. XML Toolkit 의 설치

Windows 사용자는 http://sourceforge.net/projects/pyxml 에서 패키지를, Mac 사용자는 XXX 에서 패키지를 찾을수있다. Linux 사용자는 데비안를 사용한다면 XXX 에서, RPM 을 사용한다면 http://sourceforge.net/projects/pyxml 에서 다운받을 수 있을것이다. Unix 플랫폼상에서 소스를 컴파일 하기 위해서는 다음의 설치순서를 따라하면 될것이다.

1. 만일 Python 1.5 버전을 사용한다면, 먼저 distutils 을 설치해야한다. 이것은 http://www.python.org/sigs/distutils-sig에서 찾을 수 있다. Python 1.6 이후의 버전에서는 distutils 이 포함되어 있으므로, 이 단계는 건너뛰어도 된다.
2. http://sourceforge.net/projects/pyxml에서 소스를 다운받아 다음과 같이 압축을 푼다.

   ```
   gzip -dc xml-package.tgz | tar -xvf -
   ```

3. 다음을 실행한다.:

   ```
   python setup.py install
   ```

   이 명령을 실행하기 위해서는, 적절한 C 컴파일러가 필요하며, Unix 시스템에서 슈퍼유저의 권한으로 이를 실행하여야 할것이다. setup.py 는 몇가지 서로다른 명령과 옵션을 제공하는데, 이에 대해서는 아무런 옵션없이 setup.py 를 실행하면 도움말이 나올것이다.

만일 설치에 어려움이 있다면, <xml-sig@python.org (mailto:xml-sig@python.org)>로 문제를 적어 보내주기 바란다. 또는 http://sourceforget.net/projects/pyxml로 버그리포팅을 해주어도 된다.

배포되는 소스의 demo/ 디렉토리에 많은 데모 프로그램들이 있다. 이 프로그램들을 통해, XML 툴로 가능한 일이 무엇인지 살펴보거나, 예제 소스들을 얻을 수 있을것이다.

2.1. 관련 홈페이지

http://www.python.org/topics/xml/

파이썬의 XML 토픽으로 모든 소프트웨어, 메일링 리스트, 문서등의 업데이트들을 살펴볼 수 있을것이다.

3. SAX: The Simple API for XML

SAX 는 xml-dev 메일링 리스트를 통한 많은 사람들의 조언을 통해, David Megginson 이 개발하였다. SAX 는 XML 의 파싱에 대해 이벤트-구동(event-driven) 인터페이스를 갖는다. SAX 를 사용하기 위해서, 적절한 인터페이스가 선언된 파이썬 클래스 인스턴스를 하나 생성하고, 이 오브젝트의 적절한 메쏘드를 호출함으로서 파싱을 하게 된다.

이전 버전의 문서에서는 SAX1 을 설명하였지만, 이번 하우투에서는 SAX 버전 2 (SAX2) 를 가지고 설명을 한다.

SAX 는 처음부터 끝까지 구성된 XML 문서를 읽어들여, 다른 문서로 변환하기위해 연산하거나, 그 문서의 정보들을 정리하는 (예를들어 특정 요소의 평균값등을 계산하는 작업) 등에 적합하다. 그러나, 간단한 요소의 내용이나 속성을 다른값으로 바꾸는 등의 간단한 목적에 사용될수는 있어도, 중첩된 어떠한 요소들을 교환한다던지 하는 복잡한 연산등의 문서구조 변환에는 적합하지 않다. 예를 들어, SAX 를 사용하여 속성값이 'greek'인 어떤 요소의 내용을 Greek 문자로 변환하는 등의 작업에는 적합하지만, 전체 책의 챕터를 재배치한다는 등의 작업은 적합하지 않다.

SAX 의 장점은 속도와 단순함에 있다. 만화책들을 나열하기 위한 DTD 가 정의되어 있다고 가정하고, 당신이 당신이 소장하고 있는 만화책중에서 저자(writer)가 Neil Gaiman 인 모든것을 찾으려 한다고 하자. 이러한 특정한 일에 대해서, 검색에 관련이 없는 미술가(artist)나 편집자(editor), 채색가(colourist) 등의 요소에 대한 설명을 확장할 필요는 없다. 그러므로 이럴때에는 요소중 저자(writer)를 제외한 나머지 요소는 무시하도록 클래스 인스턴스를 작성하면 된다.

또다른 이점은, 특정시점에 있어 모든 문서의 내용을 메모리에 올리지 않아도 된다는 것이다. 이것은 매우 큰 문서를 처리할때 매우 유용할것이다.

SAX 는 4개의 기본적인 인터페이스를 가지고 있다. SAX 를 따르는 파서는 데이터를 처리하기 위해 다양한 메쏘드를 호출하며, 이러한 인터페이스를 지원하는 어떠한 오브젝트도 넘겨받을 수 있다. 그러므로, 당신의 작업에 당신의 어플리케이션과 관련된 인터페이스를 이용하면 될것이다.

SAX 인터페이스는 다음과 같다.:

Interface	Purpose
ContentHandler	이 인터페이스는 SAX 의 핵심으로, 일반적인 문서 이벤트를 위한 호출이다. 이 메쏘드는 문서의
DTDHandler	기본적인 파싱에 있어 요구되는 DTD 이벤트를 핸들링하기 위해 호출된다. 즉 표기법(XML spec s
EntityResolver	외부 엔티티를 참조하기 위하여 사용된다.만일 문서에 외부 엔티티 참조가 없다면, 이 인터페이스
ErrorHandler	에러를 처리하기 위해 호출한다.파서는 모든 경고와 에러를 보고하기 위해 이 인터페이스를 통해

파이썬은 인터페이스에 대한 내용을 지원하지 않으므로 위의 인터페이스는 파이썬 클래스로 재정의 되어야 한다. 기본적으로 메쏘드는 아무런 일도 하지 않는다.(메쏘드의 내용은 파이썬의 pass 명령으로 되어있다.) 그러므로 어플리케이션에서 사용하지 않는 메쏘드에 대해서는 단순히 무시해 버리면 된다.

여기 SAX 를 사용하는 슈도코드의 예가 있다 :

```
# 핸들러 클래스의 정의
from xml.sax import Contenthandler, ...
class docHandler(ContentHandler):
    ...

# 핸들러 클래스의 인스턴스 생성
dh = docHandler()

# XML parser 생성
parser = ...

# 파서에 핸들러 인스턴스를 지정
parser.setContentHandler(dh)

# 파일 파싱; 핸들러 메소드가 호출될것이다.
parser.parse(sys.stdin)
```

3.1. Starting Out

만화책들에 대한 정보를 저장하는 간단한 XML 포맷에 대해 생각해보자. 한권에 대한 간단한 문서가 여기 있다. :

```
<collection>
  <comic title="Sandman" number='62'>
    <writer>Neil Gaiman</writer>
    <penciller pages='1-9,18-24'>Glyn Dillon</penciller>
    <penciller pages="10-17">Charles Vess</penciller>
  </comic>
</collection>
```

XML 문서는 반드시 하나의 최상위 요소를 가지고 있어야한다. 여기서는 "collection" 이 최상위 요소이다. 이것은 하나의 자식 요소인 comic 을 갖는다. comic 요소의 속성으로 책의 제목(title)과 번호(number)가 주어졌으며, 저자(writer)와 미술가(artist)에 대해 하나 또는 그이상의 자식을 가지고 있다. 하나의 간행물(issue)에 대해 여러명의 미술가(artist) 나 저자(writer)가 있을수 있다.

간단한 상태로 시작을 해보자 : FindIssue 라 이름 지어진 문서핸들러(document handler)는 주어진 간행물이 컬랙션 내에 있는지 찾는일을 할것이다.

```
from xml.sax import saxutils

class FindIssue(saxutils.DefaultHandler):
    def __init__(self, title, number):
        self.search_title, self.search_number = title, number
```

DefaultHandler 클래스는 네개의 인터페이스 - ContentHandler, DTDHandler, EntityResolver, 그리고 ErrorHandler 에서 상속을 받는다. 이것은 모든것을 하나의 클래스 내에서 사용하길 원할때 사용하는 방법이다. 만일 각각의 목적에 따라 클래스를 나누길 원하거나, 하나의 인터페이스만을 선언하길 원한다면, 각 인터페이스에 따라 각각 서브클래스를 만들수 있다. 이중에 어느 방법이 더 좋다고 할수는 없으며, 무엇을 할것인가에 따라 적절한 방법을 택하면 될것이다.

클래스가 검색을 하기 위해서, 인스턴스는 무엇을 검색할것인지를 알아야한다. 찾기를 원하는 title 과 발행물 번호는 FindIssue 생성자에게 건내지고, 인스턴스의 일부로 저장될것이다.

자 이제 실제로 동작을 하는 함수를 살펴보도록 하자. 이 간단한 작업은 단지 주어진 요소의 속성을 살펴보는 것이므로 startElement 메소드가 적합할 것이다.

```
def startElement(self, name, attrs):
    # comic 요소가 아니면 무시하라.
    if name != 'comic': return

    # title 과 number 속성에 대해 검색을 한다.
    title = attrs.get('title', None)
    number = attrs.get('number', None)
    if title == self.search_title and number == self.search_number:
        print title, '#'+str(number), 'found'
```

startElement() 메쏘드에는 주어진 요소의 이름과 그 요소의 속성을 갖는 인스턴스가 전해진다. 마지막 AttributeList 인터페이스는 파이썬 사전형이다. 그러므로 함수는 먼저 comic 요소인지 확인하고, 특정 title 과 number 속성을 검색하게 된다. 비교에 성공하면 메세지가 출력된다.

startElement() 는 문서의 모든 요소를 만날때마다 호출된다. startElement() 함수의 처음에 'Starting element:' 를 출력하도록 코드를 집어넣는다면, 다음과 같은 출력을 보게 될것이다.

```
Starting element: collection
Starting element: comic
Starting element: writer
Starting element: penciller
Starting element: penciller
```

실제 클래스를 사용하기 위해서, parser 와 FindIssue 인스턴스를 생성하고 이들을 연결하고, 입력값을 파싱하도록 파서를 호출하는 최상위 코드가 필요하다.

```
from xml.sax import make_parser
from xml.sax.handler import feature_namespaces

if __name__ == '__main__':
    # 파서의 생성
    parser = make_parser()
    # 파서에게 XML 이름영역은 무시하도록 전한다.
    parser.setFeature(feature_namespaces, 0)
```

```
# handler 의 생성
dh = FindIssue('Sandman', '62')

# 파서에 우리가 만든 핸들러를 사용하도록 전한다.
parser.setContentHandler(dh)

# 입력을 파싱한다.
parser.parse(file)
```

make_parser 클래스는 파서를 만드는 작업을 자동으로 해준다. 파이썬에는 이미 여러가지 XML 파서가 존재하며, 앞으로도 많은 파서가 추가될 예정이다. 파이썬 1.5 버전에 있는 xmllib.py 는 특별히 빠르지는 않지만, 현재도 사용가능하다. xmllib.py 의 속도를 개선한 버전이 xml.parsers 에 포함되어있다. xml.parsers.expat 모듈은 지금까지는 가장 빠르며, 가능하다면 이것이 선택된다. make_parser 는 어떤 파서가 사용가능한지를 결정하고, 가장 빠른 파서를 선택한다. 그러므로, 각 파서들이 어떻게 다른지에 대해서는 사용자가 알고 있어야 할 필요가 없다. (make_parser 에 당신이 사용하기를 원하는 파서 리스트를 알려줄수도 있다.)

SAX2 에서부터, XML 이름영역(namespace)이 지원된다. 만일 이름영역 프로세싱이 엑티브되어있다면, startElement 대신 startElementNS 이 호출된다. content 핸들러가 이름영역에 대한 메소드를 정의하지 않았다면, 이름영역 처리를 하지 않겠다고 요구하는 것이다. 이 설정의 기본은 파서에서 파서로 수정을 가한다. 그러므로 안전한 값으로 설정을 하는것이 좋다.

파서 인스턴스를 생성한 후, setContentHandler 를 호출하여 파서에게 사용할 핸들러를 넘겨준다.

예제 XML 문서를 위의 코드에 대입하면, 출력으로 'Sandman #62 found' 가 나올것이다.

3.2. Error Handling

이제 위의 코드에 아래의 파일을 입력으로 넣어보자:

```
<collection>
  &foo;
  <comic title="Sandman" number='62'>
</collection>
```

&foo; 엔티티는 알려지지 않은것이며, comic 요소는 닫히지 않았다. (만일 빈 내용을 갖는다 하더라도 ">" 로 닫기 전에 "/" 를 사용하여야 한다.) 이러한 결과로, SAXParseException 이 발생한다.

```
xml.sax._exceptions.SAXParseException: undefined entity at None:2:2
```

ErrorHandler 인터페이스의 기본코드는 어떠한 에러에 대해서도 자동적으로 예외를 발생시킨다; 만일 이러한 에러처리를 원한다면, 에러 핸들러를 수정할 필요가 없을것이다. 또한, 자기 자신만의 ErrorHandler 인터페이스나, error(), fatalError() 에 대한 최소한의 오버라이드를

만들수도 있다. 각 메소드는 최소로 한 줄 일수도 있다. warning, error, fatalError 에 대한 ErrorHandler 인터페이스 메쏘드는 모두 하나의 인자값 - exception 인스턴스를 갖는다. exception 은 항상 SAXException 의 서브클래스이며, str() 를 이용하여 읽을수있는 문제점을 설명하는 에러 메세지로 만들수있다.

다음은, 여러가지 에러발생 상황을 재설정하기 위해, 세가지중 하나의 메소드를 예외를 출력하도록 정의하였다:

```
def error(self, exception):
    import sys
    sys.stderr.write("\%s\n" \% exception)
```

이 정의에 의하면, 치명적이지 않은 error 는 에러 메세지를 출력하게 되지만, 치명적인 에러(fatal error) 는 트레이스백(traceback)을 생성하는 일을 계속할 것이다.

3.3. Searching Element Content

특정 저자(author) 가 쓴 모든 발행물(issue) 를 출력하는, 조금더 복잡한 작업을 해보자. Let's tackle a slightly more complicated task, printing out all issues written by a certain author. 이것은 저자의 이름이 <writer>Peter Milligan</writer> 과 같이 writer 요소의 내부에 존재하므로, 요소의 내용을 검색해야한다.

이러한 검색은 다음과 같은 알고리즘을 따라 처리된다:

1. startElement 메쏘드는 조금더 복잡하게 된다. comic 요소에 대해, 검색기준에 적합한 comic 을 나중에 찾기위해, 핸들러는 title 과 number 를 저장하여야한다. writer 요소에 대해서는, inWriterContent 플래그를 참으로 설정하고, writerName 속성을 빈 문자열로 설정한다.
2. XML 태그밖의 문자들(Characters)을 처리하여야 한다. inWriterContent 가 참이면, 이 문자들을 writeName 문자열에 추가한다.
3. writer 요소가 끝나면, writerName 속성안의 모든 요소의 내용을 수집한것이 된다. 그러므로, 찾고자하는 name 과 동일한것인지 검사하고, 만일 맞는다면, 이 comic 에 대한 정보를 출력한다. 다시 inWriterContent 를 거짓으로 설정한다.

여기에 첫번째 부분의 코드가 있다.

```
from xml.sax import ContentHandler
import string

def normalize_whitespace(text):
    "문자열에서 중복적으로 나타나는 공백문자(whitespace)를 제거한다."
    return string.join(string.split(text), ' ')

class FindWriter(ContentHandler):
```

```
    def __init__(self, search_name):
        # 찾고자 하는 이름을 저장한다.
        self.search_name = normalize_whitespace(search_name)

        # 플래그를 거짓(false)으로 초기화한다.
        self.inWriterContent = 0

    def startElement(self, name, attrs):
        # comic 요소이면, title 과 issue 를 저장한다.
        if name == 'comic':
            title = normalize_whitespace(attrs.get('title', ""))
            number = normalize_whitespace(attrs.get('number', ""))
            self.this_title = title
            self.this_number = number

        # writer 요소의 시작이면, 플래그를 설정한다.
        elif name == 'writer':
            self.inWriterContent = 1
            self.writerName = ""
```

startElement() 메소드에 대해서는 이전에 설명하였다. 이제 요소의 내용을 어떻게 처리하는지 보자.

normalize_whitespace() 함수는 매우 중요하므로, 코드내에 이것을 사용하여야 한다. XML 은 공백문자(whitespace) 에 매우 민감하다; 당신은 당신이 원하는 곳에 여분의 공백과 개행문자들을 포함시킬 수 있다. 이것은 속성값이나 요소의 내용을 비교하기 전에, 공백문자를 정상적으로 만들어야 함을 의미한다; 만일 두개의 요소의 내용이 다른 수의 공백문자를 갖는다면, 비교연산은 잘못된 결과를 만들어낼 것이다.

```
    def characters(self, ch):
        if self.inWriterContent:
            self.writerName = self.writerName + ch
```

characters() 메쏘드는 XML 태그밖의 문자들을 만날때 호출된다. ch 는 문자들의 스트링이지만, 반드시 바이트 스트링인것은 아니다; 파서는 문서의 일부분인 버퍼 오브젝트를 제공하거나, 유니코드 오브젝트를 건내주게 된다(파이썬 2.0 의 expat 파서). parsers may also provide a buffer object that is a slice of the full document, or they may pass Unicode objects (as the expat parser does in Python 2.0).

모든 문자들이 한번의 함수 호출로 처리된다고 가정하지 말기 바란다. 위의 예를 들어보면, "Peter Milligan" 라는 문자열에 대해 한번의 characters() 호출이 있거나, 또는 각 문자에 대해 한번씩 characters() 의 호출이 있을 수 있다. 더욱 현실적으로, 만일 내용이 "Wagner & Seagle" 과 같이 엔티티 참조를 가지고 있다면, 파서는 "Wagner ", 엔티티 참조를 상징하는 "&", 그리고 " Seagle" 에 대해 각 세번의 메소드 호출을 하게 된다.

FindWriter 의 두번째 과정에서, characters() 는 단지 inWriterContent 를 검사하고, 이 값이 참이면, 문자를 문자열에 추가하는 작업을 한다.

마지막으로, writer 요소가 끝날때, 전체 이름이 수집될것이고, 이것을 우리가 찾고자하는 이름과 비교하게 된다.

```
def endElement(self, name):
    if name == 'writer':
        self.inWriterContent = 0
        self.writerName = normalize_whitespace(self.writerName)
        if self.search_name == self.writerName:
            print 'Found:', self.this_title, self.this_number
```

다른 공백문자수에 의한 혼동을 막기위해, normalize_whitespace() 함수가 호출되었다. 이것은 이 DTD 내에서는, 이 요소가 가지고 있는 공백문자가 중요하지 않기 때문이다.

마침 태그(End tags)는 속성을 가질수 없으므로, 여기에는 attrs 인자가 없다. "<arc name="Season of Mists"/>" 와 같이 속성을 갖는 빈 태그는, startElement() 를 호출한 후 바로 endElement() 를 호출한다.

XXX 외부 엔티티 처리는 어떻게 할것인가? 그들에 대한 처리가 특별히 필요한가?

3.4. 관련 홈페이지

http://www.megginson.com/SAX/

SAX 홈페이지다. 여기에는 가장 최신의 명세서와 다양한 언어와 플랫폼상의 SAX 도구 리스트가 있다. 현재는 자바 중심적으로 되어있다.

4. DOM: The Document Object Model

문서 객체 모델(DOM)은 XML 문서를 트리구조 표현으로 명시한다. 최상위 문서 인스턴스는 트리의 루트이며, 하나의 최상위 요소 인스턴스(Element instance)를 갖는다.; 이 요소는 내용을 표현하는 자식노드나, 추가적인 자식을 갖을 수 있는 하위요소를 갖는다. 함수들은 결과 트리를 탐색하거나, 요소와 속성값으로의 접근, 노드의 추가와 삭제, 트리를 다시 XML 로 변환하는 등의 작업등을 제공한다.

DOM 은, DOM 트리를 만들어 새로운 노드를 추가하거나 하위트리를 옮긴후, 새로운 XML 문서로 결과를 출력할수있게 함으로서, XML 문서의 수정을 쉽게 해준다. 또한 자신만의 DOM 트리를 만들고, 이를 XML 로 변환하는것도 가능하다; 이 방법은 단순히 파일에 <tag1>...</tag1> 를 쓰는 방법보다, XML 을 생성하는데 더욱 유연한 방법이다.

DOM 은 한시점에서 모든 트리를 메모리에 상주시키는것을 요구하지 않는 반면에, 파이썬 DOM 도구는 RAM 에 모든 트리를 가지고 있어야 한다. 트리의 대부분을 디스크나 데이터베이스에 집어넣고, 필요한 섹션만을 읽어들이는 방법이 가능하겠지만, 현재로서는 지원되지

않는다. 이는 용량이 큰 문서를 DOM 트리로 처리할때, 메모리가 부족할 수도 있음을 나타낸다. 반대로, SAX 핸들러는 가용한 RAM 보다 큰 데이터를 처리하는것이 가능하다.

4.1. Getting A DOM Tree

DOM 트리를 얻는 가장 쉬운 방법은 자기자신이 만드는것이다. PyXML 은 DOM 을 위한 도구로서 xml.dom.minidom 과 4DOM 을 제공한다. xml.dom.minidom 은 파이썬 2.0 에 포함되어 있다. 파이썬 2.0 에 포함되어 있는 xml.dom.minidom 은 최소한의 도구로서, DOM 표준에서 요구하는 모든 인터페이스와 연산을 제공하지 않는다. 4DOM (XXX reference) 은 DOM Level 2 에 충실한 도구이므로, 이번 예제에는 이것을 사용하도록 한다.

xml.dom.package 내의 모듈중 xml.dom.ext.reader.Sax2 는 입력(파일류의 객체, 스트링, 파일이름, URL 등)에서 DOM 트리를 만들어주는, FromXmlStream, FromXml, FromXmlFile, 그리고 FromXmlUrl 함수를 제공한다. 이들은 모두 DOM 문서 객체를 반환한다.

```
import sys
from xml.dom.ext.reader.Sax import FromXmlStream
from xml.dom.ext import PrettyPrint

# parse the document
doc = FromXmlStream(sys.stdin)
```

4.2. Manipulating The Tree

DOM 은 수많은 인터페이스와 메쏘드를 가지고 있으므로, 이 HOWTO 에서 그 모든것을 설명할 수는 없다. 그러나, 다행스럽게도, DOM 권고안은 매우 가독성이 높은 문서이므로, 가능한 인터페이스에 대한 완전한 그림을 얻기 위해 이것을 읽기를 권한다. 여기서는 일부분의 개관만을 보도록 한다.

문서 객체 모델(DOM)은 XML 문서를, 노드 클래스의 하위 클래스의 인스턴스로 표현되는, 노드들의 트리로 표현한다. 어떠한 노드의 하위클래스는 요소(Element), 텍스트, 주석이 될수도 있다.

우리는 이 섹션내에서 하나의 간단한 예제문서를 사용할 것이다. 여기에 그 예제가 있다.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xbel>
  <?processing instruction?>
  <desc>No description</desc>
  <folder>
    <title>XML bookmarks</title>
    <bookmark href="http://www.python.org/sigs/xml-sig/" >
      <title>SIG for XML Processing in Python</title>
    </bookmark>
  </folder>
```

```
</xbel>
```

DOM 트리로 변환하면, 이 문서는 다음과 같은 트리로 출력된다 :

```
Element xbel None
   Text #text '  \012  '
   ProcessingInstruction processing 'instruction'
   Text #text '\012  '
   Element desc None
      Text #text 'No description'
   Text #text '\012  '
   Element folder None
      Text #text '\012    '
      Element title None
         Text #text 'XML bookmarks'
      Text #text '\012    '
      Element bookmark None
         Text #text '\012      '
         Element title None
            Text #text 'SIG for XML Processing in Python'
         Text #text '\012    '
      Text #text '\012  '
   Text #text '\012'
```

이것은 유일하게 가능한 트리는 아니다. 각기 다른 파서들이 어떻게 텍스트 노드들을 처리하느냐에 따라 달라질수 있다. 위의 트리는 텍스트 노드를 어떻게 처리하느냐에 따라 다중의 노드로 나뉘어질수도 있다.

4.2.1. The Node class

기본적인 노드 클래스를 살펴보기로 하자. 모든 Document, Element, Text 등의 기타 DOM 노드들은 노드의 서브 클래스이다. 단지 노드가 제공하는 인터페이스를 사용하여 많은 작업을 하는것이 가능하다.

XXX table of attributes and methods readonly attribute DOMString nodeName; attribute DOMString nodeValue; // raises(DOMException) on setting // raises(DOMException) on retrieval readonly attribute unsigned short nodeType; readonly attribute Node parentNode; readonly attribute NodeList childNodes; readonly attribute Node firstChild; readonly attribute Node lastChild; readonly attribute Node previousSibling; readonly attribute Node nextSibling; readonly attribute NamedNodeMap attributes; readonly attribute Document ownerDocument;

Node insertBefore(in Node newChild, in Node refChild) raises(DOMException); Node replaceChild(in Node newChild, in Node oldChild) raises(DOMException); Node removeChild(in Node oldChild) raises(DOMException); Node appendChild(in Node newChild) raises(DOMException); boolean hasChildNodes(); Node cloneNode(in boolean deep);

4.2.2. Document, Element, and Text nodes

전체 트리의 기본은 도큐먼트 노드이며, 도큐먼트 노드의 documentElement 속성은 루트 요소에 대한 요소노드(element node) 를 포함한다. 도큐먼트 노드는 ProcessingInstruction 노드와 같은 추가적인 자식을 가질 수 있다.

the complete list of children XXX.

4.3. Walking Over The Entire Tree

xml.dom.package 는 트리를 탐색(walk)하는 등의 작업에 다양한 도움을 주는 클래스를 포함하고 있다.

The Walker class

Introduction to the walker class

4.4. Building A Document

Intro to builder

4.5. Processing HTML

Intro to HTML builder

4.6. 관련 홈페이지

http://www.w3.org/DOM/

월드 와이드 웹 컨소시움의 DOM 페이지

http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/

DOM Level 1 권고안. 대부분의 표준과 달리, 이것은 코어 XML 인터페이스에만 관심이 있더라도 읽을수있게 되어있다.

5. 용어해설

XML 은 많은 용어들을 가지고 있다. 이번장에서는 그중에서 중요한 용어와 그에 관련된 내용을 설명하도록 한다.

여기에 나온 정의들은 Lars Marius Garshol 의 SGML 용어해설
(http://www.stud.ifi.uio.no/~larsga/download/diverse/sgmlglos.html
(http://www.stud.ifi.uio.no/~larsga/download/diverse/sgmlglos.html)) 에서 참고하였다.

- DOM (Document Object Model)

  문서 오브젝트 모델은 플랫폼 과 언어에 중립적인 인터페이스로서, 프로그램과 스크립트가 동적으로 문서의 내용과 구조, 스타일에 접근하고 갱신할수 있도록 해준다. 문서는 횡단하고 수정할수있는 트리구조로 재표현될 수 있다.

- DTD (Document Type Definition)

  문서 형을 정의하는 DTD 는 XML 문서형과, 완전한 요소의 타입, 엔티티 그리고 XML 선언을 정의한다. 다르게 표현한다면, DTD 는 XML 문서같은 특정한 종류의 문서를 완벽하게 묘사한다.

- SAX (Simple API for XML)

  SAX 는 xml-dev 메일링리스트의 많은 기고자에 의해 개발된 XML 파서를 위한 간단한 표준 API 이다. 인터페이스는 객체지향 언어인 경우에 한해, 언어-독립적이다. 최초에는 Java 로 구현되었지만, 현재로서는 Python 의 사용도 가능하다. SAX 는 많은 XML 파서가 지원한다.

- XML (eXtensible Markup Language)

  XML 은 개발중인 linking 과 stylesheets 에 대한 자신만의 표준을 갖으며, web 에서 사용가능한 SGML 어플리케이션의 특정 측면이다.

- XSL (eXtensible Style Language)

  XSL 은 XML 의 스타일시트 언어로서의 제안이다. 이것은 XML 문서를 매력적인 방법으로 브라우징 할수있게 하며, XML 문서를 HTML 로 변환하는 방법을 제공한다.