XML and Python Tutorial

By Alexandre Fayolle

Logilab

(mailto:alexandre.fayolle@logilab.fr)

Presentation Contents

Chapter 1 Introductions

Introducing the speaker,
his company and the tutorial…

About me

- My name is Alexandre, I'm French
- I've been using Python for the past 3 years
- Main programming interests
  - knowledge representation
  - AI algorithms
- Other interests include
  - Playing and listening Jazz
  - Writing short stories
- I've been using and constributing to PyXML and 4Suite for the past 2 years
- I maintain some Debian packages
  - python-xml, python-4suite, python-unit

About Logilab

- Logilab is a french IT company, founded in 2000
- We provide industrial-strength solution to complex problems
  - We use techniques coming from research labs and universities
  - And we mix them with good software development practices
- We are strongly involved in the Free Software and Python communities
  - Narval, python Logic-SIG, HMM, XMLDiff, PyReverse...
- **Nicolas Chauvat** is the Track Champion for the *Python in Science and Industry track at EPC2002*, together with **Marc Poinot** from ONERA
- We have some demos available on our booth over there

About this tutorial

This is not an XML tutorial:

- I won't be going in the gory syntax of XML, Schemas and DTDs

This is not a Python tutorial:

- I expect you to know the Python programming language
- I also expect that you have a good background in OO Design and Programming (Design Patterns...)

The **main focus will be on the tools** provided by the Python Standard Library and the PyXML extensions, with some mentions of the 4Suite processing tools provided by FourThought, Inc.

All the examples will use these tools.

XML syntax in 2 minutes

```
<?xml version='1.0'
encoding='iso-8859-1?>
<!DOCTYPE presentation SYSTEM
"presentation.dtd">
<presentation
xmlns="http://logilab.fr/slides">
 <info>
  <title>XML and Python
Tutorial</title>
  <author>Alexandre Fayolle</author>
 </info>
 <section title="Introductions">
  <slide title="About me">
   <list>
    <entry>My name is
<em>Alexandre</em>, I'm French</entry>
    <entry>I've been using Python for
the past 3 years</entry>
   </list>
  </slide>
  <!-- add more slides here -->
 </section>
</presentation>
```

Prologue is optional

Document Type Definition can be used to validate the document

One root element

Namespaces can be used to avoid name clashes

Elements are properly nested

Attributes must be quoted

Data is stored in attribute values or in text nodes

It's possible to have mixed contents

Some definitions

- Well formed: denotes an XML document that is syntactically correct according to the XML specification
- Valid: denotes an XML document with elements and attributes conforming to a grammar. This grammar can be described with a DTD, an XML Schema, or some other mean
- Entity: a shortcut to some data
- PCDATA: parsed character data
- CDATA: character data

Chapter 2
What is XML Processing?

Operations to be performed on XML documents

When to use XML?

- Whenever you have to decide on a format that should be open, and may have to evolve
  - Configuration files (Jabber, Zope3)
  - Document serialization (DIA, Gnumeric)
  - Business to business applications (Jabber)
- Storing data on disk
  - Human readable, structured files
  - Better extension possibilities than with *.ini* files, or *rfc822*-ish files
- Exchanging messages with remote systems
  - E.g. SOAP, XML-RPC
- Designing a pivot format for conversions
  - DocBook for technical documentation
  - XBEL for bookmarks

When not to use XML?

- XML is text-based
  - Handling binary files is not a strong point, though a binary payload is possible with external entities
  - Precision problems with handling floating point numbers
- XML is quite verbose, and can be difficult to hand-edit
  - Few good XML editors
- Parsing XML can be CPU expensive
  - Beware of possible performance problems if you have lots of parsing in a tight inner loop
  - However, there are a lot of parsers available, with various prices, licenses and performances

XML-related tasks

- There are a few number of very common programming tasks when dealing with XML data:
  - Reading (*parsing*) the data
  - Extracting some special nodes from the data
  - Changing the contents of an XML document
  - Writing the data

W3C Standards

- XML
- Document Type Definitions (DTDs)
- XML Schemas
- Document Object Model
- XPath
- XSLT
- XLink, XPointer
- RDF
- And also
  - DAML, SVG, MathML, XHTML, Xforms, etc.

Other standards

- Parsing
  - SAX
  - Lots of specific parsers (e.g. *expat, RXP*…)
- Schema languages
  - Schematron
  - REXX
  - Relax NG

Encoding considerations

- XML Data is encoded as UTF-8 (Unicode) by default
  - Or UTF-16 if a special byte sequence is included at the start of the file
- Some tools will accept other encodings if it is declared in the XML Prologue
- Parsing will generally return Unicode objects
  - Your application should be able to deal with it
  - Python's default behaviour (converting to ASCII) will work for English-speaking users
  - It will fail as soon as an 8-bit character is encountered

Encoding example

You have an XML dictionary, and want to provide a text based interface to query it.

```
<?xml version='1.0' encoding='iso-8859-1'?>
<dictionary>
 <entry>
  <word lang='en'>summer</word>
  <word lang='fr'>été</word>
 </entry>
</dictionary>
```

Trying to print the French word for 'summer' will cause the following exception:

```
UnicodeError: ASCII encoding error: ordinal not in range(128)
```

You have to find out the local encoding scheme and use the `encode(encoding)` method of the unicode objects explicitely.

Writing XML

- Generating XML is not difficult
  - You can do it easily with print statements
- Mind the encoding!
  - You have to use UTF-8 or specify the encoding in the prologue
- Mind entities!
  - Some characters must be escaped so that the output is well-formed XML

`&`	`&amp;`
`<`	`&lt;`
`>`	`&gt;` (at least in `]]>`)
`'`	`&apos;` (in attributes quoted with `'`)
`"`	`&quot;` (in attributes quoted with `"`)

Writing XML example

```
def escape(str):
    # you can also use
    # from xml.sax.saxutils import escape
    # Caution: you have to escape '&' first!
    str = str.replace(u'&',u'&amp;')
    str = str.replace(u'<',u'&lt;')
    str = str.replace(u'>',u'&gt;')
    return str

def list_as_xml(aList,aFile):
    """aList is a list of Unicode objects"""
    aFile.write('<list>\n')
    for elem in aList:
        aFile.write('<entry>')
        aFile.write(escape(elem).encode('UTF-8'))
        aFile.write('</entry>\n')
    aFile.write('</list>\n')
```

Chapter 3
Available Tools

Available Python tools
for XML processing

Standard Library

- Since version 2.0, Python comes with some excellent XML support in the standard library
  - Previous versions provided some support too, for instance the `xmllib` package in 1.5.2
- The package xml.sax and the assorted subpackages provide a non-validating SAX parser
  - Based on the Expat parser (also available with a native interface through the `xml.parsers.expat` module)
- The package `xml.dom.minidom` provides a basic dom implementation

PyXML

- The PyXML library supercedes the XML support provided by the Standard Library
  - Developed by members of the Python XML-SIG
- It adds a validating SAX parser, xmlproc
  - 100% pure python
- It adds a very compliant DOM implementation, 4DOM
  - Featuring DOM Level 2 events, readonly attributes…
  - Slow. Very slow. Uses lots of memory too
  - Loading and saving is not compatible with minidom
- 4XPath is available through the xml.xpath package
  - Works with both 4DOM and minidom
- 4XSLT is available through the xml.xslt package

4Suite

- 4Suite is a suite of tools for dealing with XML documents
  - Developed by Fourthought, Inc.
- 4DOM, 4XPath and 4XSLT used to be part of it
  - 4DOM was donated to PyXML
  - An old version of 4XPath and 4XSLT were donated too, but development goes on.
- 4Suite provides
  - enhanced 4XPath and 4XSLT
  - cDomlette, a C-based DOM implementation for Python
  - 4RDF and Versa
  - 4Suite server

Notes about PyXML/4Suite

- There are some versioning issues between PyXML and 4Suite

4Suite\PyXML	0.6.6	0.7.0	0.7.1
0.11.1	OK	NOK	NOK
0.12.0a2	NOK	NOK	OK (`xml.xslt` does't work)

Other libraries

- `PyRXP` is a very fast C parser with python bindings, released under the GPL
- `libxml` and `libxslt` are the GNOME project XML libraries with Python bindings, very fast too, available under the LGPL
- `Sablotron` is an XSLT processor with python bindings, written in C
- `Xerces` and `Xalan` are tools from the Apache XML project. There are Python bindings for the C++ versions, and the Java versions are usable from Jython

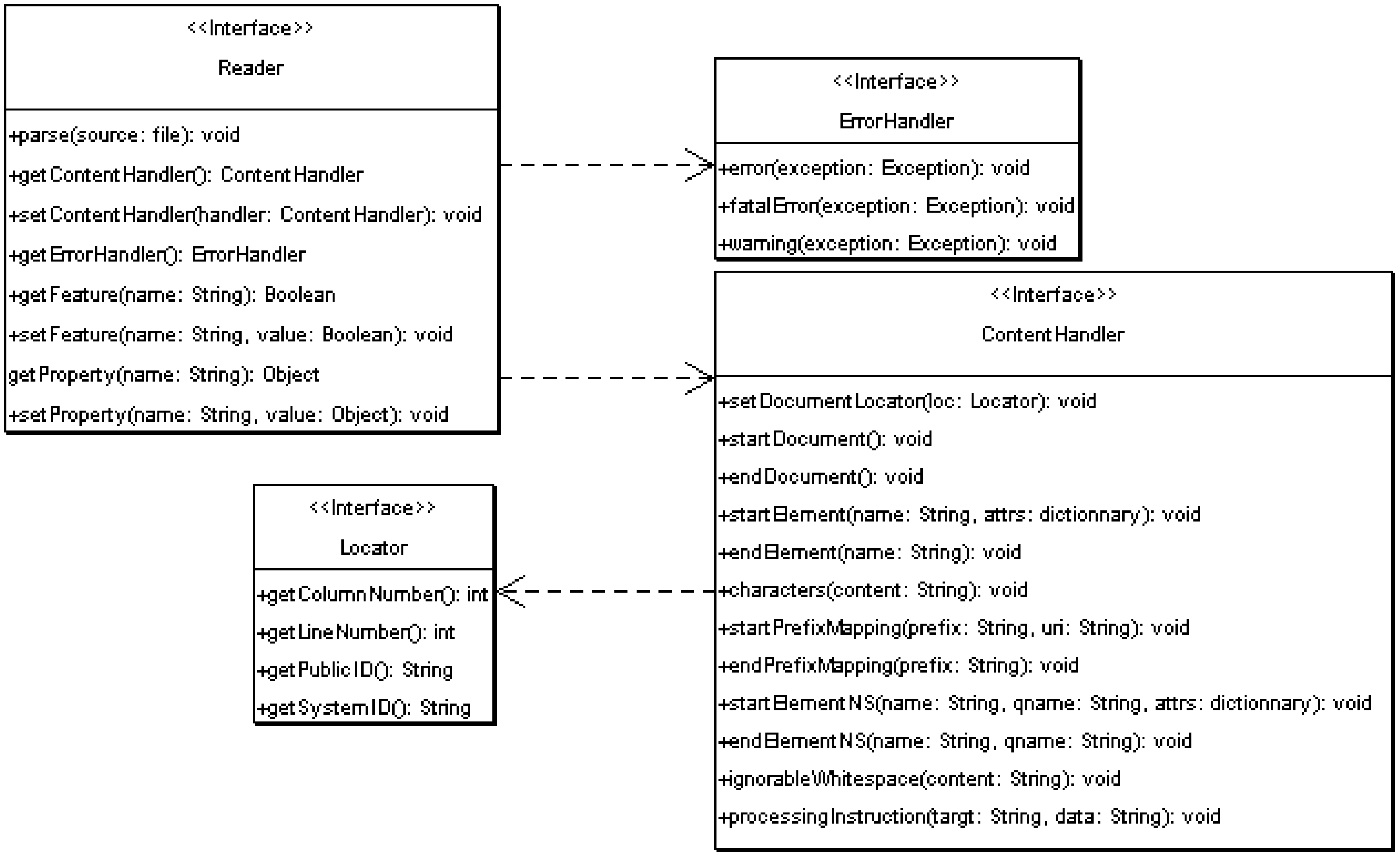
Chapter 4
Simple API for XML

Reading XML with SAX,
error handling and validation

Overall presentation

- SAX is a programming interface specifying interfaces and responsibilities for the parser and the application
  - It reduces the coupling between your application and a specific parser
- It follows the Observer design pattern
  - The library provides a class implementing the `Reader` interface
  - You provide implementations of the `ContentHandler` interface, and possibly of the `ErrorHandler` interface, and connect them to the `Reader` using the `set`*`XXX`*`Handler()` methods
  - When you call the `parse()` method, your handlers get notified of events as they are seen by the `Reader`

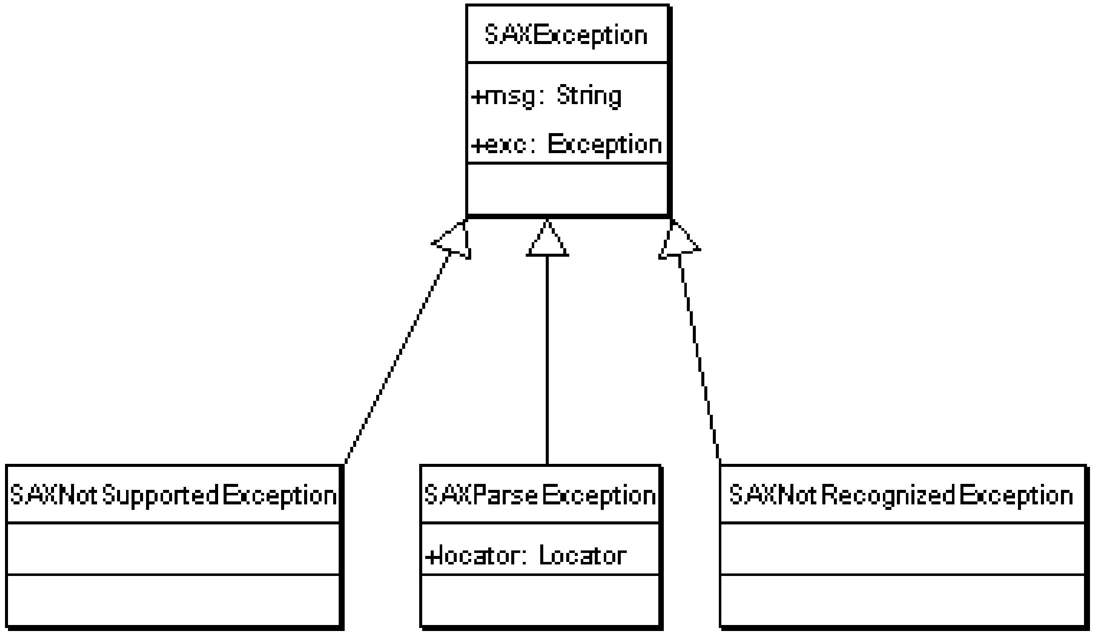
SAX main interfaces



<<Interface>>
Reader
+parse(source: file): void
+getContentHandler(): ContentHandler
+setContentHandler(handler: ContentHandler): void
+getErrorHandler(): ErrorHandler
+getFeature(name: String): Boolean
+setFeature(name: String, value: Boolean): void
getProperty(name: String): Object
+setProperty(name: String, value: Object): void
<<Interface>>
ErrorHandler
+error(exception: Exception): void
+fatalError(exception: Exception): void
+warning(exception: Exception): void
<<Interface>>
ContentHandler
+setDocumentLocator(loc: Locator): void
+startDocument(): void
+endDocument(): void
+startElement(name: String, attrs: dictionnary): void
+endElement(name: String): void
+characters(content: String): void
+startPrefixMapping(prefix: String, uri: String): void
+endPrefixMapping(prefix: String): void
+startElementNS(name: String, qname: String, attrs: dictionnary): void
+endElementNS(name: String, qname: String): void
+ignorableWhitespace(content: String): void
+processingInstruction(targt: String, data: String): void
<<Interface>>
Locator
+getColumnNumber(): int
+getLineNumber(): int
+getPublicID(): String
+getSystemID(): String

SAX Exceptions

- An instance of `SAXParseException` is passed to the `ErrorHandler` through callback method, or raised if no `ErrorHandler` was provided
- The two other exceptions are raised when dealing with unsupported or unrecognized features or properties



The Reader

- The reader is responsible for parsing the data
  - It can either do the parsing itself (xmlproc) or delegate it to a lower-level parser (expat)
  - Call the `parse(source)` method to start parsing
- It calls the methods of the various Handlers provided by applications
  - Use `setContentHandler(handler)` to set the content handler
  - Use `setErrorHandler(handler)` to set the error handler
- Its behaviour can be customized using features and properties
  - Features are boolean, properties are arbitrary objects
  - If a feature or property is not supported or not recognized trying to set it or to read its value will raise an exception

Obtaining a Reader

- Use the `make_parser()` factory function in the xml.sax package.

```
from xml.sax import make_parser
reader = make_parser()
```

- To obtain a validating reader, a little more work is required

```
from xml.sax.sax2exts import XMLValParserFactory
reader = XMLValParserFactory.make_parser()
```

  – This will return a Reader with validation capabilities and the validation feature enabled
  – If you choose to directly instanciate a validating reader, don't forget that validation is turned off by default
  – Use `setFeature(xml.sax.handler.feature_validation,1)`

Features and properties

- Feature and property names are defined in the xml.sax.handler module as symbolic constants
- We will see how to use features to enable validation and namespace processing later
- Properties can be used to add handlers besides the most common ones
  - Especially the lexical handler which can be notified of comment nodes

ContentHandler

- The `ContentHandler` is notified of parsing events by the `Reader` through its various methods

`setDocumentLocator(locator)`	First call (if ever). The locator can be asked where in the document we currently are.

`startDocument()`
`endDocument()`
	Called before the start of the root element, and after its end

`startElement(name,attributes)`
`endElement(name)`
	Called when a new element is opened or closed

`characters(string)`	Called to notify of text nodes. Be careful, since a text node can be reported with several calls to this method
`processingInstruction(name, target)`	Notifies of PI (*NB: the prologue is not reported as a PI*)

ErrorHandler

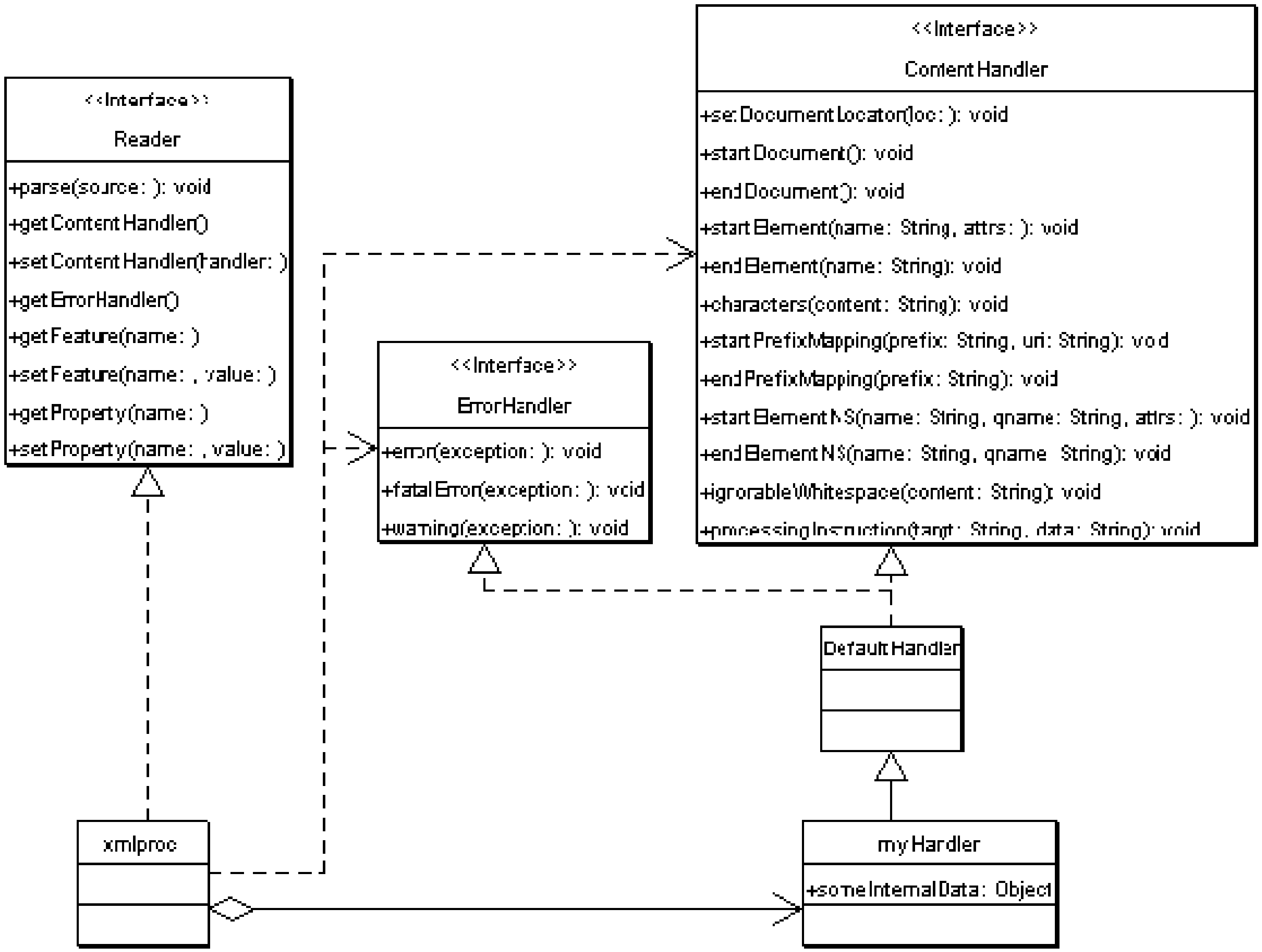
- The `ErrorHandler` is notified of parsing events by the `Reader` through its various methods

`warning(exception)`	Warnings are ignorable errors. Used, for instance for a redefined element in the DTD
`error(exception)`	Errors are triggered by non respected validation constraints
`fatalError(exception)`	Fatal errors are caused by not well formed documents

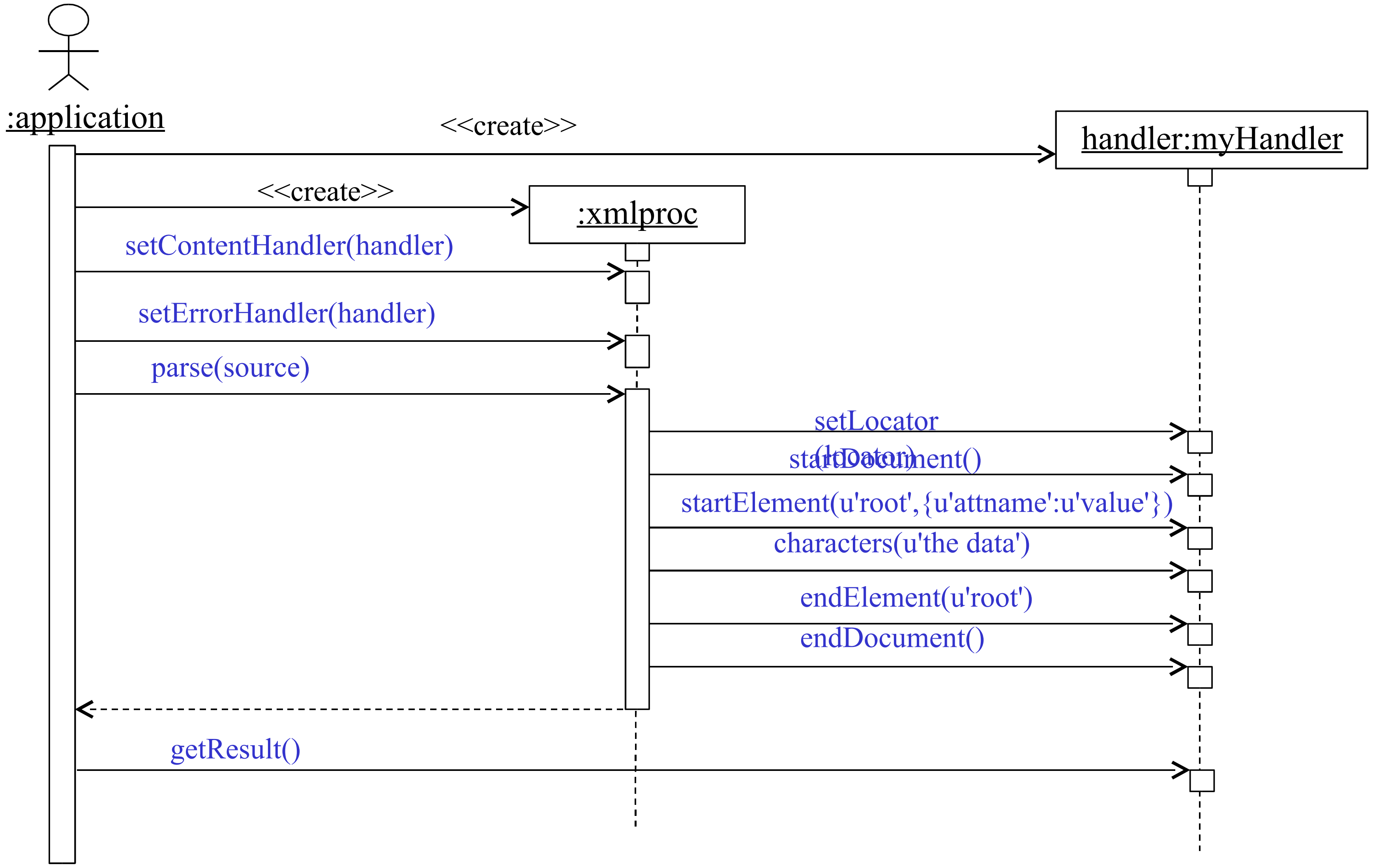
DefaultHandler

- In PyXML, a concrete implementation of the `ContentHandler` and `ErrorHandler` interfaces is provided in `xml.sax.saxutils.DefaultHandler`
- The default implementation of the callbacks do nothing
  - Except `setDocumentLocator()` which stores the locator as `self._locator`
- This is useful to start your own implementation of the handlers
  - You don't have to implement the callbacks that are not interesting for your application

Using SAX (1)


<<Interface>>
Reader
+parse(source:): void
+getContentHandler()
+setContentHandler(handler:)
+getErrorHandler()
+getFeature(name:)
+setFeature(name: , value:)
+getProperty(name:)
+setProperty(name: , value:)
<<Interface>>
ErrorHandler
+error(exception:): void
+fatalError(exception:): void
+warning(exception:): void
<<Interface>>
ContentHandler
+setDocumentLocator(loc:): void
+startDocument(): void
+endDocument(): void
+startElement(name: String, attrs:): void
+endElement(name: String): void
+characters(content: String): void
+startPrefixMapping(prefix: String, uri: String): void
+endPrefixMapping(prefix: String): void
+startElementNS(name: String, qname: String, attrs:): void
+endElementNS(name: String, qname: String): void
+ignorableWhitespace(content: String): void
+processingInstruction(target: String, data: String): void
DefaultHandler
xmlproc
myHandler
+someInternalData: Object

Using SAX (2)



:application
<<create>>
handler:myHandler
<<create>>
:xmlproc
setContentHandler(handler)
setErrorHandler(handler)
parse(source)
setLocator
(locator)
startDocument()
startElement(u'root',{u'attname':u'value'})
characters(u'the data')
endElement(u'root')
endDocument()
getResult()

Validation (1)

- Validating a document is making sure that its content follows a given grammar
  - The only support available in PyXML is DTD validation using xmlproc
- Using validation is highly recommended when loading data coming from an untrusted source
  - Anything that was read from disk or received from the network
  - You can also build your own validation system, but never make the optimistic assumption that the data you're processing has the right format

Validation (2)

- When using a validating parser, you should set up an `ErrorHandler`
  - The `error()` method of the handler will be called whenever invalid data is encountered
  - You may choose to ignore the error (for instance to report all the errors in one go at the end of the parsing)
  - Or you may raise the `SAXParseException` object passed as an argument right away, which will stop the Reader
- If you don't, the first validation error will cause an exception, and parsing will stop
- Enabling validation will cause ignorable whitespace in the document to be reported as such to the `ContentHandler`
  - through the `ignorableWhitespace()` method

Namespace processing (1)

- **Caution:** the namespace processing feature is disabled by default in the Python SAX2 binding
  - Different from the Java binding
- To enable namespace processing, use the following code:

```
from xml.sax import make_parser
reader = make_parser
from xml.sax.handler import feature_namespaces
reader.setFeature(feature_namespaces,1)
```

Namespace processing (2)

- When namespace processing is enabled, then no calls are made to startElement() and endElement()

`startPrefixMapping(prefix,nsuri)`
`endPrefixMapping(prefix)`
	Called to signify that a new prefix is used. The calls to these methods occur before and after the calls to `startElementNS` and `endElementNS` respectively.

`startElementNS(name,qname,attrs)`
`endElementNS(name,qname)`
	Name is a 2 element tuple `(uri,localname)`;
	`qname` is the qualified name or `None` if this is not supported by the parser (expat will use `None`)

Building Objects from SAX

- In this sample application, we will use a SAX parser to build Python objects.
  - Each element name will be used as a class name.
    - XML attributes will be used as named arguments to the constructor of the class
  - Sub elements objects will be added to the parent object using `addClassname(obj)` method calls
  - Text nodes will be passed using a call to `setData()`
- Note that this scheme will perform validation without using a DTD
  - The validation code is in the class definitions

Example data

```
<Window bgcolor='cyan'>
  <Panel layout='vertical'>
    <Label>Hello World</Label>
    <Button>OK</Button>
  </Panel>
</Window>
```

```
class Window:
    def __init__(fgcolor='black',
                 bgcolor='white'):
        self.fg = fgcolor
        self.bg = bgcolor
        self.content = None
    def addPanel(self,panel):
         self.content = panel
```

```
class Panel:
    def __init__(layout='horizontal'):
        self.layout = layout
        self.contents = []
    def addLabel(self,label):
         self.content.append(label)
    def addButton(self,butn):
         self.content.append(butn)
```

Add a Label and a Button class, with a `setData(d)` method

Example code (1)

```
def safe_eval(str):
    """converts a string to an int, a float or its repr"""
    try:
        return int(str)
    except:
        try:
            return float(str)
        except:
            return repr(str)
```

Example code (2)

```
from xml.sax.handler import ContentHandler
class ObjBuilder(ContentHandler):
    def __init__(self, class_names):
        ContentHandler.__init__(self)
        self.obj = []
        self._classes = class_names
        self.__buffer = []
    def startElement(self,name,attrs):
        self.__buffer = []
        if name not in self._names: raise NameError(name)
        args = [u'='.join(k,safe_eval(v) for k,v in attrs.items()]
        o = exec(u'%s(%s)'%(name,u','.join(args)))
        if self.obj: apply(self.obj[-1].getattr('add%s'%name),(o,))
        else: self.rootobject = o
        self.obj.append(o)
    def endElement(self,name):
        if self.__buffer:
            self.obj[-1].setData(u''.join(self.__buffer))
            self.__buffer = []
        del self.obj[-1]
    def characters(self,contents):
        self.__buffer.append(contents)
```

Example code (3)

```
def build_class_hierarchy(classnames,file):
    from xml.sax import parse
    handler = ObjBuilder(classnames)
    parse(file,handler)
    return handler.rootobject
```

Chapter 5
Document Object Model

Building, writing,
using a DOM

Overall presentation

- DOM is an XML element to Object mapping defined by the W3C
  - It is used a lot in browser scripting
  - Standardization for loading and saving DOMs is still nonexistent
    - Therefore, each implementation has its own way of doing this
- Each part of a XML document is mapped to an interface for which an implementation is provided by libraries
  - Reasonably portable code, apart from loading/saving
- DOM depends on a parser to build the objects in memory

DOM vs SAX

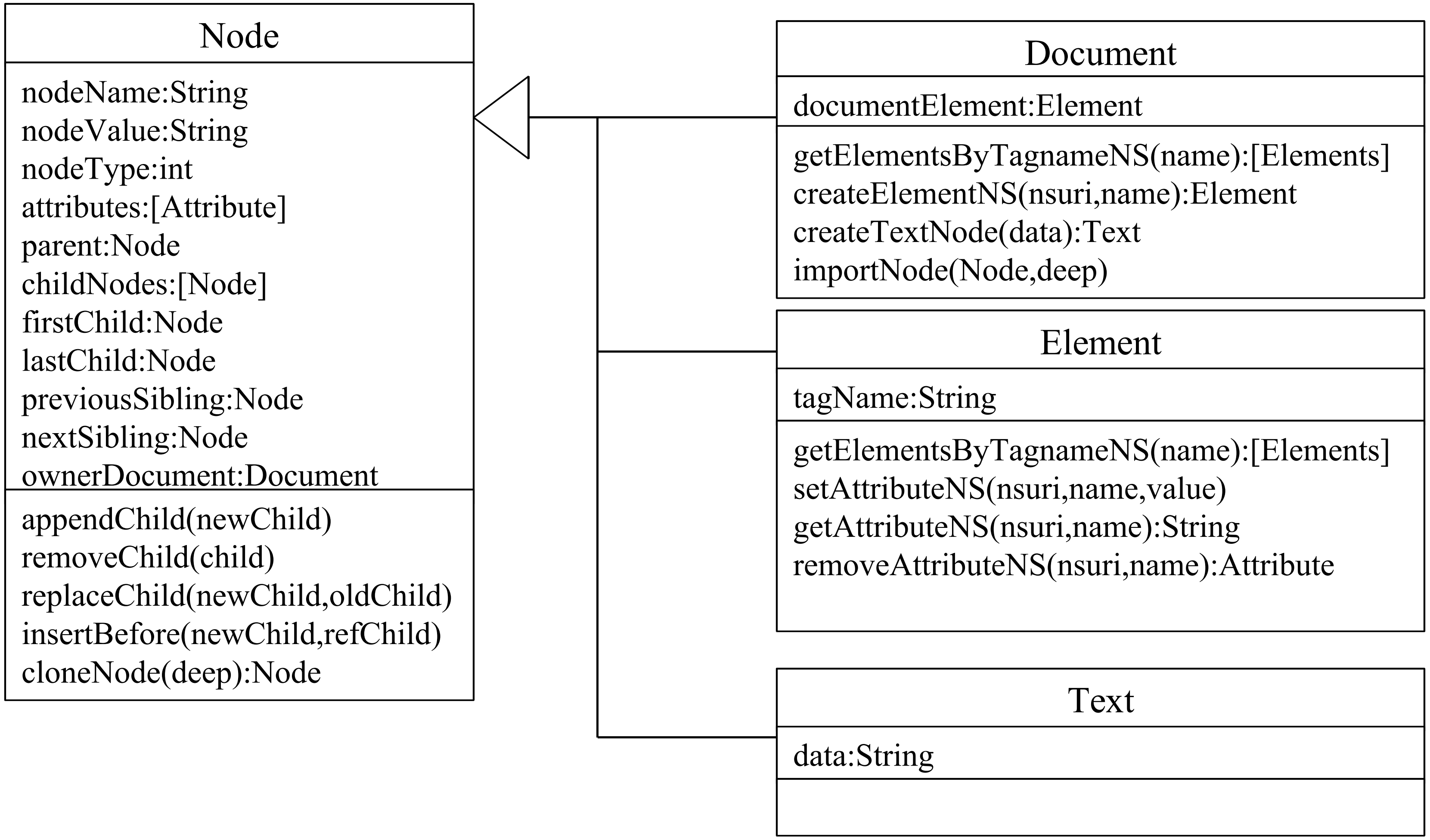
In DOM:

- Full document view
- Random navigation
- Must load the whole document first
- High memory consumption
- Can create new nodes, and move nodes in the document

In SAX:

- Document as stream
- Forward only
- Get events as soon as parsing begins
- Use only the memory you need
- Difficult to add nodes, impossible to move nodes

DOM Nodes


Node
nodeName:String
nodeValue:String
nodeType:int
attributes:[Attribute]
parent:Node
childNodes:[Node]
firstChild:Node
lastChild:Node
previousSibling:Node
nextSibling:Node
ownerDocument:Document
appendChild(newChild)
removeChild(child)
replaceChild(newChild,oldChild)
insertBefore(newChild,refChild)
cloneNode(deep):Node
Document
documentElement:Element
getElementsByTagnameNS(name):[Elements]
createElementNS(nsuri,name):Element
createTextNode(data):Text
importNode(Node,deep)
Element
tagName:String
getElementsByTagnameNS(name):[Elements]
setAttributeNS(nsuri,name,value)
getAttributeNS(nsuri,name):String
removeAttributeNS(nsuri,name):Attribute
Text
data:String

Building a DOM (1)

- This is highly implementation dependent
- For minidom, use:

```
from xml.dom.minidom import parseString, parse
doc1 = parseString(str)
doc2 = parse(file)
```

- For 4DOM, use:

```
from xml.dom.ext.reader.Sax2 import Reader
reader = Reader()
doc1 = reader.fromString(str)
doc2 = reader.fromStream(file)
doc3 = reader.fromUri(docuri)
```

Building a DOM (2)

- For Domlette in 4Suite >= 0.12.0a2, use:

```
from Ft.Xml import Domlette
reader = Domlette.NonValidatingReader
doc1 = reader.parseString(str,'<someUri>')
doc2 = reader.parseStream(open(filename),filename)
doc3 = reader.parseUri(docuri)
```

Writing a DOM

- This is highly implementation dependent
- For minidom, use:

```
from xml.dom.minidom import parseString
doc = parseString(str)
xmlstr = doc.toxml() # use toprettyxml() for pretty printing
doc.writexml(open('document.xml','w'))
```

- For 4DOM, use:

```
from xml.dom.ext.reader.Sax2 import Reader
doc = Reader().fromString(str)
from xml.dom.ext import Print, PrettyPrint
from StringIO import StringIO
strio = StringIO()
Print(doc,strio) # use PrettyPrint(…) for pretty printing
xmlstr = strio.get_value()
Print(doc,open('document.xml','w'))
```

DOM Navigation

- For each Node object (that is almost any element in a DOM), it is possible to navigate in any of the five basic directions in a tree using
  - `parent`, `previousSibling`, `nextSibling`, `firstChild`, `lastChild`
  - If no corresponding element exist, the value of the attribute is None
- You can also get to the top of the tree, using the `ownerElement` attribute
  - This is `None` if the current node is a `Document`
    - **Caution**: it is `self` in 4DOM Documents
- The `childNodes` attribute is a *live* list of all the nodes children
  - Attributes are not in the `childNodes`
  - *Live* means that it gets updated if you add or remove children

Navigation example

- Let `node` be an Element in a Document. We know that its second child with tag name 'data' holds a text node, and we want to get the data in this text node

```
def get_the_second_data(node):
    count = 0
    for c in node.childNodes:
        if c.tagName == 'data':
            count +=1
            if count == 2:
                textnode = c.firstChild
                return textnode.data
```

- **Caution:** there can be comment nodes, or text nodes full of ignorable whitespace in the childNodes list. Do not depend on node counting when navigating in a DOM

Moving and adding nodes

- To move a node, get a reference to it, and a reference to its new parent node, and just use
  - `newParent.appendChild(node)`
  - `newParentNode.insertBefore(node,ref)`
    - `ref` can be `newParent.firstChild` to insert node at the beginning
  - This will update all the affected attributes
- To create a new Node, you need a Document instance
  - Use `ownerDocument` on any node to get the document it belongs to
  - Then use one of the factory methods to create the node
  - Finally, insert it in the document

Example

```python
from xml.dom.minidom import parseString
# create a new document
doc = parseString(u'<article/>'.encode('UTF-8'))
art = doc.documentElement
# create a sect1 element
s1 = doc.createElementNS(None,u'sect1')
# add it under the root element
art.appendChild(s1)
# create a title element with a text node inside
s1.appendChild(doc.createElementNS(None,u'title'))
title = doc.createTextNode(u'Introduction to XML')
s1.firstChild.appendChild(title)
s1.appendChild(doc.createElementNS(None,u'para'))
txt = doc.createTextNode(u'WRITE ME!')
s1.lastChild.appendChild(txt)
# write the result
print doc.toprettyxml()
```

Example output

```
<?xml version="1.0" ?>
<article>
     <sect1>
          <title>Introduction to XML</title>
          <para>WRITE ME!</para>
     </sect1>
</article>
```

Working with attributes

- Attributes are best handled though the `set/getAttributeNS()` methods of the `Element` interface
  - You can also see them as nodes, but it takes more work to do the same thing
- If an attribute is not set, trying to get it will return an empty string
- Setting an attribute will overwrite its previous value, if any

Working with 2 documents

- If you want to add nodes from another Document, you have to import them first
  - This is done with a call to

    ```
    newnode = Document.importNode(node,deep=1)
    ```
  - You can then append the new node somewhere in your document

Chapter 6 XPath and XSLT

Random access to a Document, converting XML to other formats

XPath Presentation

- The W3C has defined an XML Stylesheet Language (XSL). XPath and XSLT are part of this specification
- XPath can be used to point to a node or a set of nodes in a document, or even to make some computations based on values in the document
  - Requires having a DOM representation, since the engine needs to iterate in the XML tree in an order that is different from document traversal order
- There are implementations of XPath and XSLT in pure Python available in PyXML and 4Suite, as well as several bindings to C, C++ or Java implementations

XPath syntax

- An XPath is a sequence of location steps separated by `'/'`
- Each location step selects nodes in the XML tree using the previous location step as a context
- A location step is made of
  - An optional axis, giving the navigation direction
  - A node test
  - An optional predicate, used to filter nodes passing the test

```
/addressbook/person[name]/phone[@type='mobile']/text()
```

Using 4XPath

- To use 4XPath from a recent version of PyXML, or 4Suite < 0.12, use the xml.xpath package:

```
from xml.xpath import Evaluate
```

- To use 4XPath from 4Suite >= 0.12, use the Ft.Xml.XPath package

```
from Ft.Xml.XPath import Evaluate
```

- This function takes an xpath in a string and a context node as arguments. It returns a list of nodes matching the xpath, or a value corresponding to the evaluation of the xpath

XSLT Principles

- XSL(T) is a functional language that lets you define transformation/styling rules, aka templates.
- An XSLT processor will take two inputs, a stylesheet/transform and an XML document, and output the result. The result may be text, xml or html.
- The XSLT processor will walk the tree of nodes from the input document and apply the matching templates defined in the stylesheet.
- As with any functional language, recursion is the key.
- Let's walk through an example.

Input Document

```
<addressbook>
  <person>
    <name>Eric Idle</name>
    <phone>999-999-999</phone>
    <phone type='mobile'>555-555-555</phone>
    <address>
      <street>12, spam road</street>
      <city>London</city>
      <zip>H4B 1X3</zip>
    </address>
  </person>
  <person>
  <name>Terry Gilliam</name>
    <phone type='mobile'>555-555-554</phone>
    <phone>999-999-998</phone>
    <address>
      <street>3, Brazil Lane</street>
      <city>Leeds</city>
      <zip>F2A 2S5</zip>
    </address>
  </person>
</addressbook>
```

XSL Transformation

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
              xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="text" encoding="ISO-8859-1"/>
 <xsl:template match="addressbook">
Mobile Phone Numbers From AddressBook
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  <xsl:apply-templates select="person">
   <xsl:sort select="name"/>
  </xsl:apply-templates>
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 </xsl:template>
 <xsl:template match="person">
  <xsl:value-of select="name"/>'s <xsl:apply-templates
                                     select="phone[@type='mobile']"/>
 </xsl:template>
 <xsl:template match="phone">
  <xsl:value-of select="@type"/>: <xsl:value-of select="text()"/>
  <xsl:text>
</xsl:text>
 </xsl:template>
</xsl:stylesheet>
```

Using 4XSLT (1)

- To use 4XSLT from a recent version of PyXML, or 4Suite < 0.12, use the `xml.xslt` package:

```
from xml.xslt.Processor import Processor
```

- To use 4XSLT from 4Suite >= 0.12, use the Ft.Xml.XSL package

```
from Ft.Xml.Xslt.Processor import Processor
```

- The Processor API changed, so the code depends on which version you're using
  - 4Suite 0.12.0a2 uses `InputSource`s to abstract the origin of a document
  - PyXML and previous versions of 4Suite had several methods to append a stylesheet or process a document read from a string, and file or a URI.

Using 4XSLT (2)

```
# PyXML Code
from xml.xslt.Processor import Processor
p = Processor()
p.appendStylesheetString(ssheet)
result = p.runString(idoc)
print result

# 4Suite >= 0.12.0a2 Code
from Ft.Xml.Xslt.Processor import Processor
from Ft.Xml.InputSource import DefaultFactory
p = Processor()
stylesheet_source = DefaultFactory.fromString(ssheet,
                                        'stylesheet uri')
doc_source = DefaultFactory.fromString(idoc,'source uri')
p.appendStylesheet(stylesheet_source)
result = p.runString(doc_source)
print result
```

Output

```
$ 4xslt addressbook.xml addbook.xslt
Mobile Phone Numbers From AddressBook
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Eric Idle's mobile: 555-555-555
Terry Gilliam's mobile: 555-555-554
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Chapter 7 Where can you go now?

Pointers to further readings and documentation sources

Mailing lists

- General XML-related mailing lists
  - W3C mailing lists : http://w3.org/Mail/Lists.html
  - XSL mailing list: http://www.mulberrytech.com/xsl/xsl-list/
- Python-specific mailing lists
  - XML-SIG: http://mail.python.org/mailman/listinfo/xml-sig/
  - 4Suite: http://www.fourthought.com/mailman/listinfo/4suite/

Web resources and docs

- PyXML and 4Suite:
  - http://pyxml.sf.net and http://4suite.org/
- Documentations links:
  - Collection of pointers on the XML-SIG page: http://pyxml.sf.net/topics/
  - Uche Ogbuji's integrated guide to XML processing in Python : http://uche.ogbuji.net/tech/akara/pyxml
  - Online Python cookbook: http://aspn.activestate.com/ASPN/Cookbook/Python?kwd=XML
  - Python XML Wiki: http://twistedmatrix.com/users/jh.twistd/xml-sig/moin.cgi/
- Lars Marius Garshol's book, Definitive XML Application Development

Interesting tools

- XMLDiff
  - Logilab module, GPLed
  - Computes differences between XML documents
- mxTidy
  - Python wrapper around W3C's Tidy utility
  - Converts HTML to XHTML
  - Can handle MSOffice "html" files