XML

XML

extensible Markup Language

Introduction

- This Lecture Covers
  - What is XML?
  - Tags and Content
  - Attributes
  - Nesting
  - Namespaces and Comments
  - Indentation
  - Schemas

XML

- eXtensible Markup Language
- Like HTML
  - Hypertext Markup Language
  - Used for creating websites
  - XML can be used for any kind of structured data

Caveat

- There are entire *books* written about XML
  - This lecture is not intended to tell you everything
- This will hopefully cover enough about XML to get you started understanding SOAP-based web services

Tags

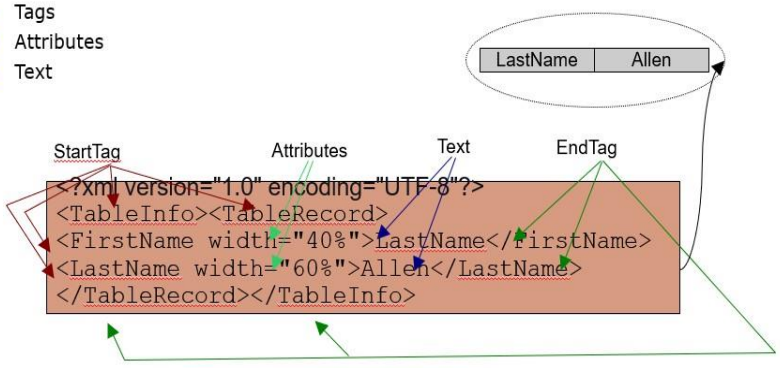
- Tags work like HTML
- Start tags have this format: <_____>
- End tags have this format: </_____>
- The start and end tags must match
- Tag names must only be letters, numbers, and underscores
- Example:
  `<artist>The Beatles</artist>`
- An empty tag can also end with />
  - Example:
  `<invitees/>` is the same as `<invitees></invitees>`

Structure of an XML document

- The eXtensible Markup Language (XML) is a W3C recommendation for creating Special-purpose markup languages (information format) that enable the structuring, description and interchange of data.

Basic elements:

- Tags
- Attributes
- Text



Content

- What goes between the tags is content
- If content has no tags:
  - Treated like a string, even though it has no quotes
  - You can use it for numbers and true/false
    - The software will convert from a string
- If content has tags:
  - Nested data

Nested Tags

- You can put tags inside of tags to create nested data
- Example:

```
<color>
  <red>205</red>
  <green>123</green>
  <blue>52</blue>
</color>
```

Attributes

- In addition to content, tags can have attributes
- Attributes hold simple data (a string)
- Attributes are key/value pairs.
- Both are strings, but the keys do not have quotes
  - This means they must be letters, numbers, and underscores only
  - No spaces or punctuation characters
- Attributes appear in the start tag
  - key="value"

Attributes, continued

- In the most common designs, attributes are not used for data
- They are used to indicate that some property about the data (metadata)
- Examples:

```
<fileSize unit="kB">34.6</fileSize>

<cost currency="USD">43.23</cost>

<projectedValue decimals="2" confidence="5">
    23.45</projectedValue>
```

The first line of an XML file

- XML declaration
  - Indicates that it's an XML file
  - Says what version, character encoding, etc.
  - XML - declaration is not a tag. It is used for the transmission of the meta-data of a document.
- Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

XML Example

- Business card

John Smith
+1 (415) 555-1234 (home)
+1 (800) 555-9867 (work)
+1 (510) 555-1212 (mobile)
john@smith.com

XML Example

- Business card: xml applies **description** to the **data** that you are working with

<BusinessCard>
<name> John Smith </name>
<phone type="home">+1 (415) 555-1234 </phone>
<phone type="work">+1 (800) 555-9867 </phone>
<phone type="mobile">+1 (510) 555-1212 </phone>
<email> john@smith.com</email>
</BusinessCard>

Example XML: Describing a song

```
<?xml version="1.0" encoding="UTF-8" ?>
<song>
   <title>Hey Jude</title>
   <artist>The Beatles</artist>
   <musicians>
       <musician>John Lennon</musician>
       <musician>Paul McCartney</musician>
       <musician>George Harrison</musician>
       <musician>Ringo Starr</musician>
   </musicians>
</song>
```

The same XML with attributes

```
<?xml version="1.0" encoding="UTF-8" ?>
<song title="Hey Jude" artist="The Beatles">
   <musicians>
       <musician name="John Lennon"/>
       <musician name="Paul McCartney"/>
       <musician name="George Harrison"/>
       <musician name="Ringo Starr"/>
   </musicians>
</song>
```

Namespaces

- You might have a tag name mean one thing in one context, but something else in a different context
- For this, you use namespaces to uniquely identify them
- The tag name has a "namespace: " prefix
- Example:

```
<sdkbridge:onlineCourse>
```

Well Formedness

XML checked for *well-formedness*.

Most tags have to be closed – you can't be as sloppy as with HTML.

"Empty" tags not enclosing look like this: `<TAG />` or `<TAG/>`.

Case-sensitive.

Schema

As well as checking for well-formedness we can check whether a document is *valid* against a *schema* : definition of the specific XML type.

There are two popular schema types in XML:

(older) DTD (Document Type Definition)

(newer) XSD (XML Schema Definition)

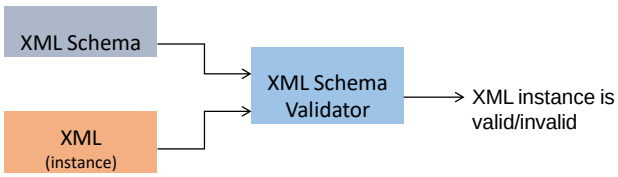
XSD more complex, but is XML itself – only need one parser.

In a separate text file, linked by a URI (URL or relative file location).

Schemas

- XML files can be described with a schema
- Called a XSD file (XML Schema Definition)
- Describes what the tags, attributes, and types are
  - XSD files are in XML
  - It's pretty easy to figure out how they work
- They can be helpful in documenting XML

Validate XML docs against XML Schema



Simple example

```
<?xml version="1.0" encoding="UTF-8"?>

<map>
  <polygon id="p1">
      <points>100,100 200,100 200,
                  200 100,000 100,100</points>
  </polygon>
</map>
```

XSD Schema

```
<xsi:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.geog.leeds.ac.uk"
  xmlns="http://www.geog.leeds.ac.uk"
  elementFormDefault="qualified">
<xsi:element name="map">
    <xsi:complexType>
        <xsi:sequence>
            <xsi:element name="polygon" minOccurs="0" maxOccurs="unbounded">
                <xsi:complexType>
                    <xsi:sequence>
                        <xsi:element name="points" type="xsi:string"/>
                    </xsi:sequence>
                    <xsi:attribute name="id" type="xsi:ID"/>
                </xsi:complexType>
            </xsi:element>
        </xsi:sequence>
    </xsi:complexType>
</xsi:element>
</xsi:schema>
```

XML Schema for Book

<**sequence**> indicator means that the child elements must appear in the same order as they are declared.

A **complex element** is an XML element that contains other elements and/or attributes.

Restrictions are used to define acceptable values for XML elements or attributes

Read more: https://www.w3schools.com/xml/schema_intro.asp

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="Book">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Title" type="xs:string"/>
                <xs:element name="Authors">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="Author"
type="xs:string" maxOccurs="5"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="Date" type="xs:gYear"/>
                <xs:element name="Publisher" minOccurs="0">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="Springer"/>
                            <xs:enumeration value="MIT Press"/>
                            <xs:enumeration value="Harvard Press"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

XSD 1.1 also provides 3 built-in datatypes, "gYear", "gMonth" and "gDay", to cover 3 properties within a "date" value with these rules

XML Schema for Book

The <**xs:all**> element is used to specify that child elements can appear in any order and each child element can occur at most once. So, it cannot be used when having a repeating element such as <Author>, as it can appear more than once inside <Authors>.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="Book">
        <xs:complexType>
            <xs:all>
                <xs:element name="Title" type="xs:string"/>
                <xs:element name="Authors">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="Author"
type="xs:string" maxOccurs="5"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="Date" type="xs:gYear"/>
                <xs:element name="Publisher" minOccurs="0">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="Springer"/>
                            <xs:enumeration value="MIT Press"/>
                            <xs:enumeration value="Harvard Press"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
            </xs:all>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Use the following validator to test it out: https://www.freeformatter.com/xml-validator-xsd.html

XML Schema for Book

In your example, the Authors complex type is defined to contain a sequence of author elements, each of which contains a **name** element. This complex type is then used as the type of the Authors element within the Book element declaration.

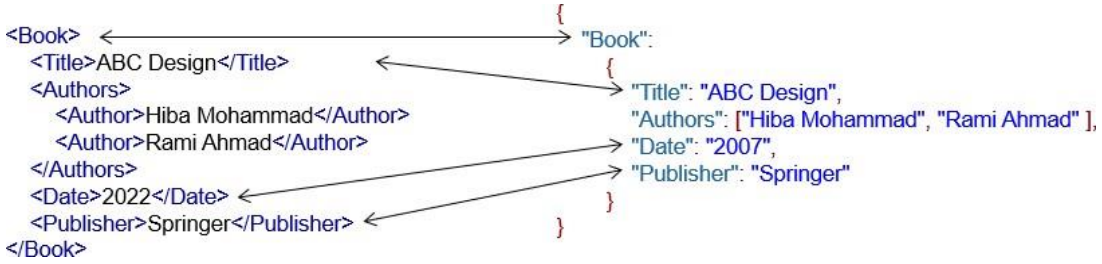
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="Book">
        <xs:complexType>
            <xs:all>
                <xs:element name="Title" type="xs:string"/>
                <xs:element name="Authors" type="Authors" />
                <xs:element name="Date" type="xs:gYear"/>
                <xs:element name="Publisher" minOccurs="0">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="Springer"/>
                            <xs:enumeration value="MIT Press"/>
                            <xs:enumeration value="Harvard Press"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
            </xs:all>
        </xs:complexType>
    </xs:element>

  <xs:complexType name="Authors">
    <xs:sequence>
      <xs:element name="author" type="Author" maxOccurs="5"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Author">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Defining a complexType allows you to specify the structure of a complex element. Once you have defined a complexType, you can use it as the type of another element by referencing its name in the type attribute of the element declaration. This allows you to reuse the complex type definition in multiple places within your schema.

XML and JSON, side-by-side



AJAX

- Asynchronous JavaScript and XML (AJAX)
  - A group of interrelated web development techniques used on the client-side to create interactive web applications
  - Web apps can fetch data from the server without refreshing the page
- AJAX is used to increase interactivity and dynamism of web pages

Multiple views and Transformation

Nice thing is that this data can be styled in lots of different ways using *stylesheets*.

To write these, we use the *XSL (eXtensible Stylesheet Language)*.

This has several parts, two of which are *XSLT (XSL Transformations)* and *XPath*.

XPath

Allows you to navigate around a document.

For example:

"/." : root of the document.

"@" : an attribute.

"//" : all elements like this in the XML.

`/.p/h2` – all 2nd-level headers in paragraphs in the root

`/.p/h2[3]` – 3rd 2nd-level header in paragraphs in the root

`//p/h2` – all 2nd-level headers in any paragraph.

`//p/h2[@id="titleheader"]` – all 2nd-level headers in any paragraph where id=titleheader.

Numerous build-in functions for string, boolean, and number operations.

XSLT

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method='html' version='1.0' encoding='UTF-8' indent='yes'/>

<xsl:template match="/.">
<html>
        <body>
                <h2>Polygons</h2>
                <p>
                <xsl:for-each select="/map/polygon">
                        <P>
                                <xsl:value-of select="@id"/> :
                                <xsl:value-of select="points"/>
                        </P>
                </xsl:for-each>
                </p>
        </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Converts XML to HTML.

Linking to XSLT

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="map3.xsl"?>
<map
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.geog.leeds.ac.uk
map3.xsd"
>

<polygon id="p1">
        <points>100,100 200,100 200,
                        200 100,000 100,100</points>
</polygon>
</map>
```

Views - Result

As XML

```
<?xml version="1.0" encoding="UTF-8" ?>
- <map xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.geog.leeds.ac.uk map2.xsd">
  - <polygon id="p1">
      <points>100,100 200,100 200,200 100,000 100,100</points>
    </polygon>
</map>
```

As HTML

Polygons

p1 : 100,100 200,100 200,200 100,000 100,100

Home exercises

- How we can read/update an XML document, find two types of parsers, try to implement a simple program in Python that do the following:

1. Read and update an XML document. This document should contain a list of students, each with their
   name, ID, date of birth, and semester average.
2. Write an XSD document for the above XML document.
3. Use online validator to see how schema validation restrict the XML document data types.
4. Implements functionality in Python to input a student's ID and retrieve the respective student's information from the XML document.

XML

Styling and other issues

Python and XML

XML Parsing

Two major choices:

Document Object Model (DOM) / Tree-based Parsing:

The whole document is read in and processed into a tree-structure that you can then navigate around, either as a DOM (API defined by W3C) or bespoke API.

The whole document is loaded into memory.

Stream based Parsing:

The document is read in one element at a time, and you are given the attributes of each element.

The document is not stored in memory.

Standard library

xml library contains:

`xml.etree.ElementTree`	: parse to tree
`xml.dom`	: parse to DOM
`xml.dom.minidom`	: lightweight parse to DOM
`xml.sax`	: SAX push and pull parser
`xml.parsers.expat`	: SAX-like push and pull parser
`xml.dom.pulldom`	: pull in partial DOM trees

Other libraries

`lxml` : simple XML parsing

Can be used with SAX (http://lxml.de/sax.html) but here we'll look at simple tree-based parsing.

Validation using lxml

Against DTD:

```
dtd_file = open("map1.dtd")
xml1 = open("map1.xml").read()
dtd = etree.DTD(dtd_file)
root = etree.XML(xml1)
print(dtd.validate(root))
```

Against XSD:

```
xsd_file = open("map2.xsd")
xml2 = open("map2.xml").read()
xsd = etree.XMLSchema(etree.parse(xsd_file))
root = etree.XML(xml2)
print(xsd.validate(root))
```

Note extra step of parsing the XSD XML

Parsing XML using lxml

```
root = etree.XML(xml1)              # Where xml1 is XML text
print (root.tag)                    # "map"
print (root[0].tag)                 # "polygon"
print (root[0].get("id"))           # "p1"
print (root[0][0].tag)              # "points"
print (root[0][0].text)             # "100,100 200,100" etc.
```

```
<map>
<polygon id="p1">
        <points>100,100 200,100 200,
                200 100,000
100,100</points>
</polygon>
</map>
```

Generating XML using lxml

```
root = etree.XML(xml1)                                  # Could start from
nothing
p2 = etree.Element("polygon")                           # Create polygon
p2.set("id", "p2");                                     # Set attribute
p2.append(etree.Element("points"))                      # Append points
p2[0].text = "100,100 100,200 200,200 200,100"  # Set points text
root.append(p2)                                         # Append polygon
print (root[1].tag)                                     # Check
```

Transform XML

```
xsl3 = open("map3.xsl").read()          # Read stylesheet
xslt_root = etree.XML(xsl3)                     # Parse stylesheet
transform = etree.XSLT(xslt_root)               # Make transform
result_tree = transform(root)                   # Transform some XML
root
transformed_text = str(result_tree)

print(transformed_text)
writer = open('map3.html', 'w')         # Normal writer
writer.write(transformed_text)
```

Note that if the XML is from a file it doesn't need the XSL is referenced in the XML, a major advantage in applying arbitrary stylesheets.

Other libraries

`dicttoxml` : conversion of dicts to XML
`untangle` : library for converting DOMs to object models

Not distributed with Anaconda, but worth looking at.
Nice intro by Kenneth Reitz at:
http://docs.python-guide.org/en/latest/scenarios/xml/

Review

- XML represents structured data
- Content is enclosed in tags
- Attributes are key/value pairs, usually with metadata
- Tags can be nested – use indentation for formatting
- Namespaces can help make tag names unique
- XSD (schema) files describe the XML structure