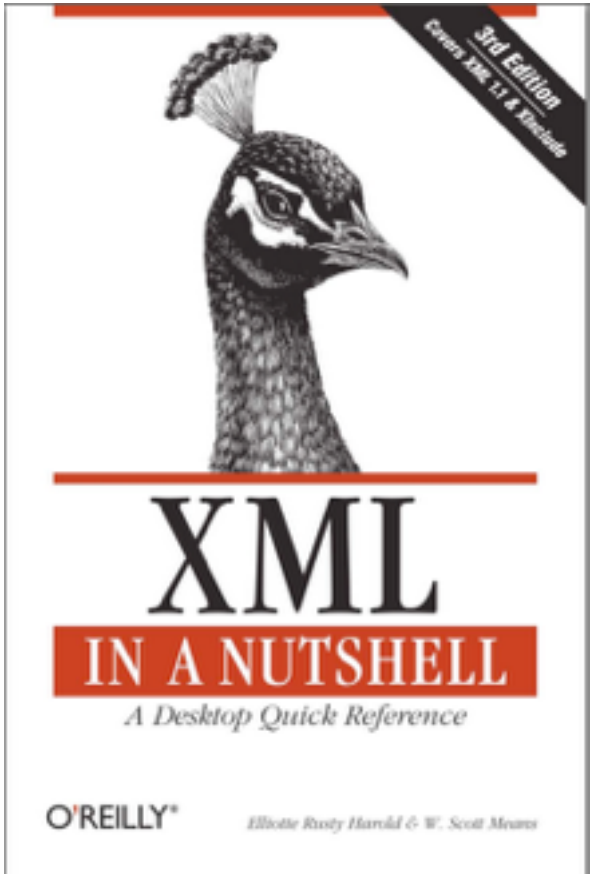
CS 2316 Data Manipulation
XML

Christopher Simpkins
chris.simpkins@gatech.edu

Extensible Markup Language (XML)

- A W3C standard for document markup
- Used for web pages, data interchange, configuration files, remote procedure calls, object serialization, ...
- All examples in this lecture come directly from, or are adapted from O'Reilly's XML in a Nutshell



XML in a Nutshell, 3ed, Elliotte Rusty Harold and W. Scott Means

Benefits of XML

- Human-readable plain text
  - Caveat: XML primarily for machines to read and write
- Extensible: XML is a metamarkup language in which you can define custom tag sets for your domain
  - Customized tag sets are called XML *applications*
- XML tag sets can be designed to represent semantics, with presentation separated into style sheets
- With a schema or DTD, XML documents can be machine validated, greatly facilitating interoperability
  - Validity includes well-formedness, completeness, data type consistency
- XML is NOT a programming language

Simple XML Example

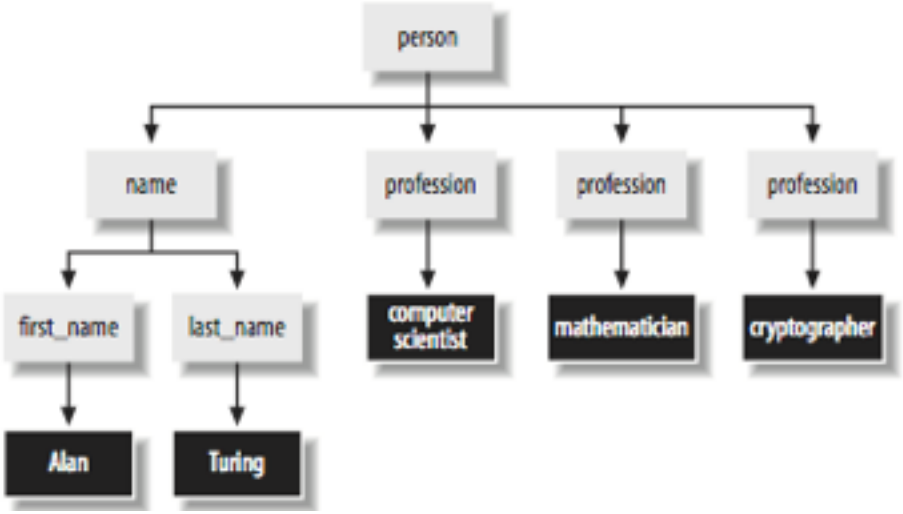
```
<person>
  Alan Turing
</person>
```

This is a simple, yet complete XML document. It could be stored in a file called person.xml, or be stored in a file that contains several XML documents, or be automatically generated from a database query.

- Elements are marked up with tags.
- Here, `<person>` is start tag and `</person>` is end tag, which denote the person element.
- The character data between the start and end tags are the element's content
- Tags are case-sensitive `<person>` != `<Person>`
- An empty person element could be written `<person></person>` or `<person/>`

XML Tree Structure

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```



- Nesting elements create a tree structure
- Every XML document has a document root, which in this example is `person`
- `person` is the *parent* of the name element and the three profession elements, which are *children* of the `person` element

Mixed Content

```
<biography>
  <paragraph>
    <name><first_name>Alan</first_name> <last_name>Turing</last_name>
    </name> was one of the first people to truly deserve the name
    <emphasize>computer scientist</emphasize>. Although his
    contributions to the field are too numerous to list, his
    best-known are theeponymous <emphasize>Turing Test</emphasize> and
    <emphasize>Turing Machine</emphasize>.
  </paragraph>

  <definition>The <term>Turing Test</term> is to this day the standard
  test for determining whether a computer is truly intelligent. This
  test has yet to be passed. </definition>

  <definition>A <term>Turing Machine</term> is an abstract finite
  state automaton with infinite memory that can be proven equivalent
  to any other finite state automaton with arbitrarily large
  memory. Thus what is true for one Turing machine is true for all
  Turing machines no matter how implemented. </definition>

  <paragraph>
    <name><last_name>Turing</last_name></name> was also an
    accomplished <profession>mathematician</profession> and
    <profession>cryptographer</profession>. His assistance was crucial
    in helping the Allies decode the German Enigma cipher. He
    committed suicide on <date><month>June</month> <day>7</day>,
    <year>1954</year></date> after being convicted of homosexuality
    and forced to take female hormone injections.
  </paragraph>
</biography>
```

- Previous XML example, and most we'll deal with, are record-like data structures
- Narrative documents can also be marked up with XML, which is sometimes called "mixed content"

XML Attributes

```
<person born="1912-06-23" died="1954-06-07">
  Alan Turing
</person>
```

```
<person born="1912-06-23" died="1954-06-07">
  <name first="Alan" last="Turing"/>
  <profession value="computer scientist"/>
  <profession value="mathematician"/>
  <profession value="cryptographer"/>
</person>
```

Person data modeled with attributes.

```
<length units="cm">100</length>
```

`units` is metadata, so we model it with an attribute.

- Information can be represented as attributes
- Attributes are encoded as `name="value"` pairs in an element's start tag
- Model data with elements or attributes?
  - Up to you
  - My advice (which is the standard advice): use elements for your data, attributes for metadata
  - Not often clear what's data and what's metadata

XML Names

- Elements and attributes have names
- Names may contain any alphabetic character from any language, digits, and
  - underscores (_)
  - hyphens (-)
  - periods (.)
- May not contain quotation marks, apostrophes, dollar signs, carets, percent symbols, and semicolons
- Names must start with underscore or alphabetic character

Names starting with `XML` (in any case) are reserved for W3C XML-related specs

XML Name Examples

- Acceptable names:
  - `<Drivers_License_Number>98 NY 32</Drivers_License_Number>`
  - `<month-day-year>7/23/2001</month-day-year>`
  - `<_4-lane>I-610</_4-lane>`
  - `<téléphone>011 33 91 55 27 55 27</téléphone>`
  - `<πepcha>ΓanhaoNbahob</πepcha>`
- Not acceptable names:
  - `<Driver's_License_Number>98 NY 32</Driver's_License_Number>`
  - `<month/day/year>7/23/2001</month/day/year>`
  - `<first name>Alan</first name>`
  - `<4-lane>I-610</4-lane>`

Entity References

- Angle brackets (<>) and ampersand (&) are reserved for XML syntax, so you can't include these characters in your character data (element content and attribute values)
- XML provides 5 predefined entity references:
  - `<` The less-than sign, a.k.a. the opening angle bracket (<)
  - `&` The ampersand (&)
  - `>` The greater-than sign, a.k.a. the closing angle bracket (>)
  - `&quot;` The straight, double quotation marks (")
  - `&apos;` The apostrophe, a.k.a. the straight single quote (')
- Only < and & are strictly necessary, others provided for symmetry

Entity Reference Examples

```
<SCRIPT LANGUAGE="JavaScript">
  if (location.host.toLowerCase().indexOf("ibiblio") &lt; 0) {
    location.href="http://ibiblio.org/xml/";
  }
</SCRIPT>
```

Here, the < symbol in the boolean expression is represented by the < entity reference.

```
<company>W.L. Gore &amp; Associates</company>
```

Here, the & symbol in the element content is represented with the & entity reference.

```
<company>W.L. Gore &#38; Associates</company>
```

This is equivalent to the second example above. You can use entity references to represent any Unicode 2.0 code point (38 is ampersand). Note that XML 1.1 will support Unicode 3.0.

CDATA and Comments

```
<p>
You can use a default <code>xmlns</code> attribute
to avoid havingto add the svg prefix to all your
elements:
</p>

<pre>
<![CDATA[
<svg xmlns="http://www.w3.org/2000/svg"
     width="12cm" height="10cm">
  <ellipse rx="110" ry="130" />
  <rect x="4cm" y="1cm" width="3cm" height="6cm" />
</svg>
]]>
</pre>
```

- Sometimes using entity references can be tedious, such as when your element content is an X/HTML code snippet
- A CDATA section contains raw character data surrounded by `<![CDATA[` and `]]>`
- Only character sequence disallowed in a CDATA section is `]]>`

Processing Instructions

```
<?robots index="yes" follow="no"?>
```

```
<?php
  mysql_connect("database.unc.edu", "clerk", "password");
  $result = mysql("HR", "SELECT LastName, FirstName
     FROM Employees ORDER BY LastName, FirstName");
  $i = 0;
  while ($i < mysql_numrows ($result)) {
    $fields = mysql_fetch_row($result);
    echo "<person>$fields[1] $fields[0] </person>\r\n";
    $i++;
  }
  mysql_close( );
?>
```

```
<?xml-stylesheet href="person.css" type="text/css"?>
<person>
  Alan Turing
</person>
```

- Enclosed in <? and ?>
- A way to pass information to a specific (kind of) XML processor
- Most common uses:
  - embedded programming language code
  - style sheets

XML Declaration

- Very beginning of an XML document. Optional, but should be there
- Example:
  - `<?xml version="1.0" encoding="ASCII" standalone="yes"?>`
- Three attributes:
  - Version should always be 1.0
  - Encoding is the character set. Default is UTF-8, which is a superset of ASCII (so all ASCII text is also UTF-8)
  - Standalone says whether the document can be used without a DTD. Some DTDs provide default values for optional attributes
- Version attribute is required, encoding and standalone attributes are optional

Well-Formedness

- Every start-tag must have a matching end-tag.
- Elements may nest but may not overlap.
- There must be exactly one root element.
- Attribute values must be quoted.
- An element may not have two attributes with the same name.
- Comments and processing instructions may not appear inside tags.
- No unescaped < or & signs may occur in the character data of an element or attribute.
- ... and more.

Parsing and Validation

- A parser reads an XML document and places it into a structure for processing (like a tree data structure - slide 5)
- A validating parser can validate an XML document while parsing it if given an external document that specifies the structure and content of the XML document
- All XML documents must be well-formed to be parsed, validity is optional
- Validity means conformance to a:
  - Document Type Definition (DTD)
  - XML Schema Language document (XSD)
  - RelaxNG document
  - Schematron document
- DTD and XML schema are most popular. We'll cover XML schemas

Namespaces

- An XML schema or DTD defines a *vocabulary* for XML documents, that is, a tag set
- An XML document can include tags from multiple vocabularies
  - Namespaces resolve conflicts between different vocabularies
- Namespaces are identified with URIs in the root element of an XML document. Example:
  - <song xmlns="http://songs.com/singerVocab">
- Namespaces can be given a prefix. Example:
  - <song:song xmlns="http://songs.com/singerVocab">
  - Then elements in document that use tags from song vocabulary must be prefixed with song:

Namespace Example

```
<songs>
  <song>
    <singer>
      <title>M.C.</title>
      <name>Hammer</name>
    </singer>
    <title>Hammer Time</title>
  </song>
</songs>
```

```
<song:songs xmlns:singer="http://songs.com/singerVocab"
            xmlns:song="http://songs.com/songVocab"
            elementFormDefault="qualified">

  <song:song>
    <singer:singer>
      <singer:title>M.C.</singer:title>
      <singer:name>Hammer</singer:name>
    </singer:singer>
    <song:title>Hammer Time</song:title>
  </song:song>
</song:songs>
```

- In the first example, the title element name is in conflict - it comes form both the singer vocabulary and the song vocabulary
- In the second example, the conflict is resolved with namespaces
- In the second example, all elements have qualified names

XML Schemas

- An XML Schema is an XML document containing a formal description of a valid XML document
- An XML document that conforms to a schema is said to be *schema-valid*, or simply *valid*
- A valid XML document is called an *instance* of its associated schema

Associating an XML Document with a Schema

- An XML document is associated with a schema with one of 3 possible methods in the start tag of its root element:
  - An xsi:schemaLocation attribute on an element contains a list of namespaces used within that element and the URLs of the schemas with which to vali- date elements and attributes in those namespaces.
  - An xsi:noNamespaceSchemaLocation attribute contains a URL for the schema used to validate elements that are not in any namespace.
  - A validating parser may be instructed to validate a given document against an explicitly provided schema, ignoring any hints that might be provided within the document itself.

A Simple XML Schema Example

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="fullName" type="xs:string"/>
</xs:schema>
```

The schema in address.xsd defines XML documents with one element: name

```
<?xml version="1.0"?>
<fullName>Scott Means</fullName>
```

The document in addresses.xml conforms to the address schema in address.xsd

Element Declarations

- We've already seen a simple element declaration:
  - <xs:element name="fullName" type="xs:string">
- The simple types that an element may take are:
  - anyURI - A uniform resource identifier
  - base64Binary - Base64-encoded binary data
  - boolean - true, false, 1, or 0
  - byte - A signed byte in the range [-128, 127]
  - dateTime - An absolute date and time
  - duration - A length of time in years, months, days, hours, etc.
  - ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS - defined in the attribute declaration section of the W3C XML 1.0 recommendation
  - integer
  - language - Values defined in the xml:lang attribute in the XML 1.0 recommendation
  - Name - Any XML name (see slide 8)
  - string - A Unicode string

Attribute Declarations

```
<xs:element name="fullName">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="language"
                      type="xs:language"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

A schema declaration for a fullName element with a language attribute and simple content

```
<fullName language="en">Scott Means</fullName>
```

An XML element conforming to the schema above

- Having attributes makes an element `complexType`
- Elements with simple content are declared by extending a base type, such as `xs:string`
- Extension also contains attribute declarations
- Can still have simple content, as in the example to the left

Complex Types

```
<xs:element name="fullName">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="first" type="xs:string"/>
      <xs:element name="middle" type="xs:string"
                  minOccurs="0"/>
      <xs:element name="last" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

A fullName element consisting of nested elements for first and last names. Note that middle name is optional due to minOccurs constraint of 0

```
<fullName>
  <first>Rob</first>
  <last>Myers</last>
</fullName>
```

A fullName element conforming to the schema declaration above

- Complex types can have attributes and nested elements
- Nested elements are specified as a `sequence`
- Nested elements can have occurrence constraints
  - In the example to the left, middle name is optional (`minOccurs="0"`)

A Complete Example

```xml
<?xml version="1.0"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>

        <xs:element name="fullName">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="first" type="xs:string"/>
              <xs:element name="middle" type="xs:string"
                          minOccurs="0"/>
              <xs:element name="last" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>

        <xs:element name="profession" minOccurs="1"
                    maxOccurs="unbounded"/>

      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

person.xsd

```xml
<?xml version="1.0"?>

<person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="person.xsd">
  <fullName>
    <first>Alan</first>
    <last>Turing</last>
  </fullName>
  <profession>Mathematician</profession>
  <profession>Computer Scientist</profession>
  <profession>Cryptographer</profession>
</person>
```

alanturing.xml

XML Parsing

- CSV files are simply structured text-based data files
- XML files are more highly structured text-based data files, allowing nested, or “tree” structured data to be represented in text
- An XML Parser reads an XML file and extracts its structure. Three kinds of XML parsers:
  - SAX (Simple API for XML): a state machine that produces events for each element read from an XML file. Parser responds to events to process these elements
  - DOM (Document Object Model): the DOM standard specifies an object-tree representation of an XML document
    - » Tip: In Google Chrome you can see the DOM tree of any web page by clicking View->Developer->Developer Tools in the menu and clicking the Elements tab in the lower pane. Expand elements of the DOM tree and cursor over them to see the rendered parts of the page they represent. In FireFox you can install a plug-in to do the same thing
- We’ll use ElementTree, which is essentially a Python-specific DOM parser

ElementTree

- An `ElementTree` is a representation of an XML document as a tree of `Element` objects with easy to use methods:
  - `parse("fileName")` reads an XML file and create an ElementTree representation of its document
  - `find("elementName")` gets a single child of an element
  - `findall("elementName")` gets an iterator over the like-named children of an element
- That's it. Really. It's that simple.

A Complete XML Parsing Example

- Say we have an XML file named `people.xml`

```
<?xml version="1.0"?>

<people>
  <person>
    <firstName>Alan</firstName>
    <lastName>Turing</lastName>
    <profession>Computer Scientist</profession>
  </person>
  <person>
    <firstName>Stephen</firstName>
    <lastName>Hawking</lastName>
    <profession>Physicist</profession>
  </person>
</people>
```

- Our XML document will have a root element named `people` and child elements named `person`
- Each person element will have three child elements named `firstName`, `lastName`, and `profession`

Parsing with ElementTree

- Parsing `people.xml` with `ElementTree` is this easy:

```
>>> import xml.etree.ElementTree as et
>>> people = et.parse("people.xml")
>>> persons = people.findall("person")
>>> for person in persons:
...     print(person.find("firstName").text)
...     print(person.find("lastName").text)
...     print(person.find("profession").text)
...
Alan
Turing
Computer Scientist
Stephen
Hawking
Physicist
>>>
```

Conclusion

- XML makes a great data format for information exchange
- Validation is important when passing XML data form one system to another
- You now know how to write well-formed XML
- You now know how to read a simple XML schema and write an XML document that conforms to it
- We've only introduced XML schemas; there's much more to know
- Parsing XML in Python is super easy