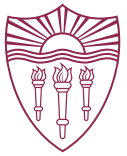
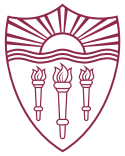


DSCI 510
PRINCIPLES OF PROGRAMMING FOR DATA SCIENCE

Itay Hen



Web Scraping

PARSING HTML

What is Web Scraping

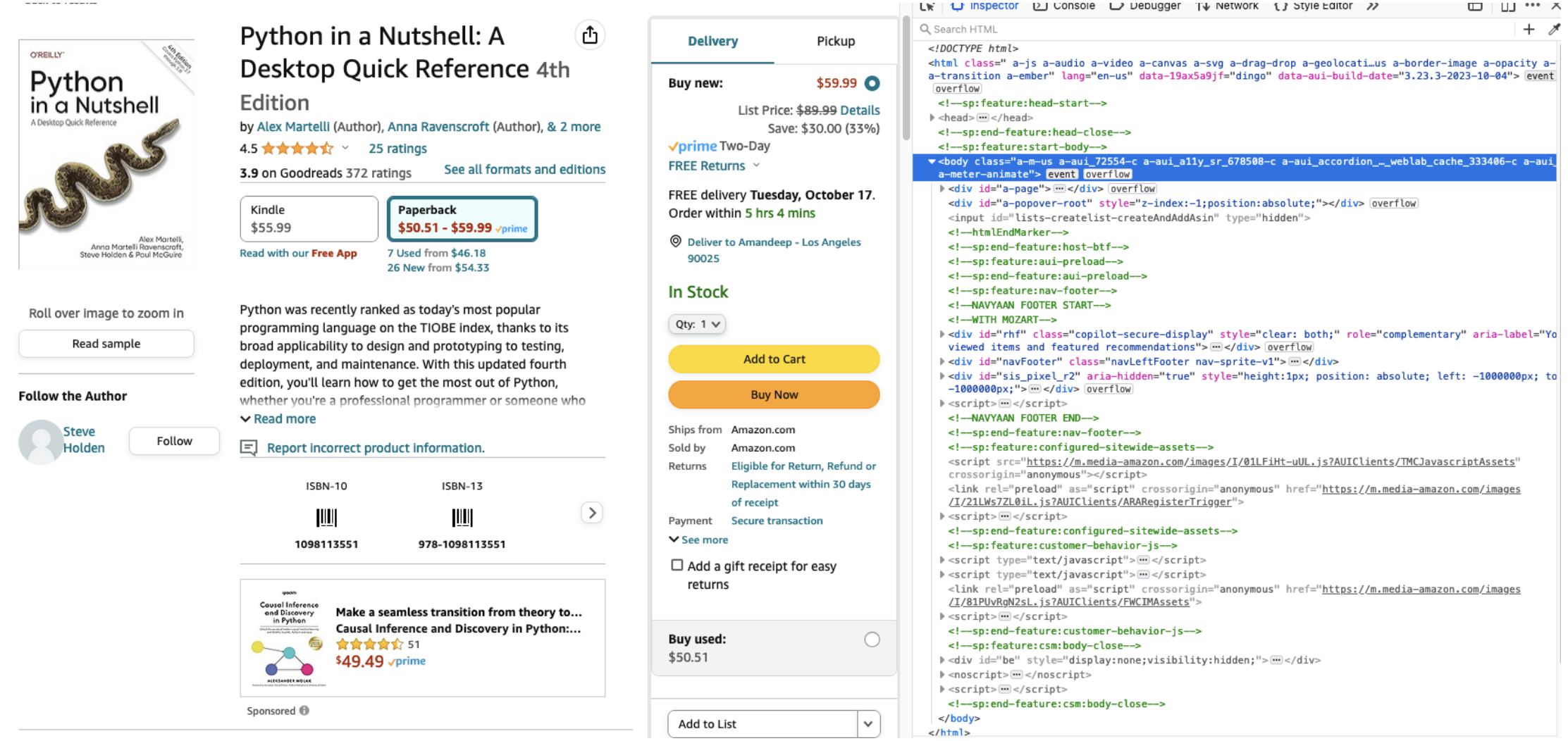
- When a program or script pretends to be a browser and retrieves web pages, looks at those web pages, extracts information, and then looks at more web pages
- Search engines scrape web pages - we call this “spidering the web” or “web crawling”
- Scraping a web page involves fetching it and extracting from it.
- Fetching is the downloading of a page (which a browser does when a user views a page).
- Once fetched, extraction can take place. The content of a page may be parsed, searched and reformatted, and its data copied into a spreadsheet or loaded into a database.

http://en.wikipedia.org/wiki/Web_scraping
http://en.wikipedia.org/wiki/Web_crawler

Why Scrape?

- Pull data - particularly social data - who links to who?
- Monitor a site for new information
- Spider the web to make a database for a search engine
- There is some controversy about web page scraping and some sites are a bit snippy about it.
  - Republishing copyrighted information is not allowed
  - Violating terms of service is not allowed
- Recall r=requests.get("…") from last week.
- We can scrape by analyzing r.text.
- But then, we'll have to interpret and parse the text ourselves.

Scraping an Amazon Page



The Easy Way: BeautifulSoup

- Documentation: https://beautiful-soup-4.readthedocs.io/en/latest/
- BeautifulSoup is a Python library for pulling data out of HTML and XML files.
- Maps an HTML string to a Document Object Model (DOM)
- Install via pip:

```
pip install bs4
```

Some BeautifulSoup Tutorials

- https://www.pythonforbeginners.com/python-on-the-web/web-scraping-with-beautifulsoup
- https://www.pythonforbeginners.com/beautifulsoup/beautifulsoup-4-python
- https://www.pluralsight.com/guides/web-scraping-with-beautiful-soup
- https://beautiful-soup-4.readthedocs.io/en/latest/

requests + BeautifulSoup

```
from bs4 import BeautifulSoup
```

- `bs4` is the **module (or package)** — it's the *Beautiful Soup 4* library.
- `BeautifulSoup` is a **class** defined *inside* that module.

```
url = 'http://www.dr-chuck.com/page1.htm'
response = requests.get(url)
```

HTTP GET method

```
if response.status_code == 200:
```

200 means OK

```
    html = response.text
    soup = BeautifulSoup(html, 'html.parser')
```

get the data using the .text method
create an instance of BeautifulSoup

```
    # Retrieve all the anchor tags
    tags = soup.findAll('a')
    for tag in tags:
        print(f'tag: {tag}')
        print(f'href: {tag.get("href", None)}')
```

```
tag: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
href: http://www.dr-chuck.com/page2.htm
```

BeautifulSoup:

Inspecting HTML Elements

```
print(soup.title)          # show <title> tag
print(soup.title.text)      # get the text within title
print(soup.p)               # first <p> tag (paragraph)
print(soup.a['href'])       # extract hyperlink reference
```

Finding Elements

```
soup.find('p').   # first <p> tag (same as soup.p)
soup.find_all('a')                          # all <a> tags
soup.find_all('div', class_='container')# divs with class 'container'
```

Reminder: Common HTML Tags

Document

<html> Creates an HTML document

<head> Sets off the title & other info that isn't displayed

<body> Sets off the visible portion of the document

<title> Puts name of the document in the title bar; when bookmarking pages, this is what is bookmarked

Structure

<h1> … <h6> : Headings -- H1=largest, H6=smallest

<p> Creates a new paragraph

 Used to format block content with CSS

 Used to format inline content with CSS

 Inserts a line break

Lists

 Creates an unordered list

 Creates an ordered list

 each list item

Links

clickable text Creates a hyperlink to a Uniform Resource Locator

clickable text Creates a hyperlink to an email address

<a name="NAME"> Creates a target location within a document

clickable text Creates a link to that target location

Text Tags

 Emphasizes a word (usually processed in bold)
<b> Creates bold text (should use instead)

 Emphasizes a word (usually processed in italics),
<I> Creates italicized text (should use instead)

<pre> Creates preformatted text
<code> Used to define source code, usually monospace,
<tt> Creates typewriter-style text

Graphical elements

 Adds image located at URL

<hr> Inserts a horizontal rule

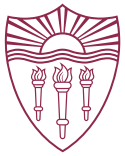
BeautifulSoup:

Navigating the DOM Tree

```
div = soup.find('div')          # find first <div>
print(div.parent)                 # access its parent
print(div.children)               # iterate children
print(div.next_sibling)           # move to next sibling
```

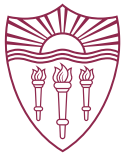
Example: Extracting Links

```
url = "https://example.com"                  # target page
r = requests.get(url)                          # fetch HTML
soup = BeautifulSoup(r.text, "html.parser")  # parse HTML
for a in soup.find_all('a', href=True):  # iterate over all links
    print(a['href'])                           # display href attribute
```



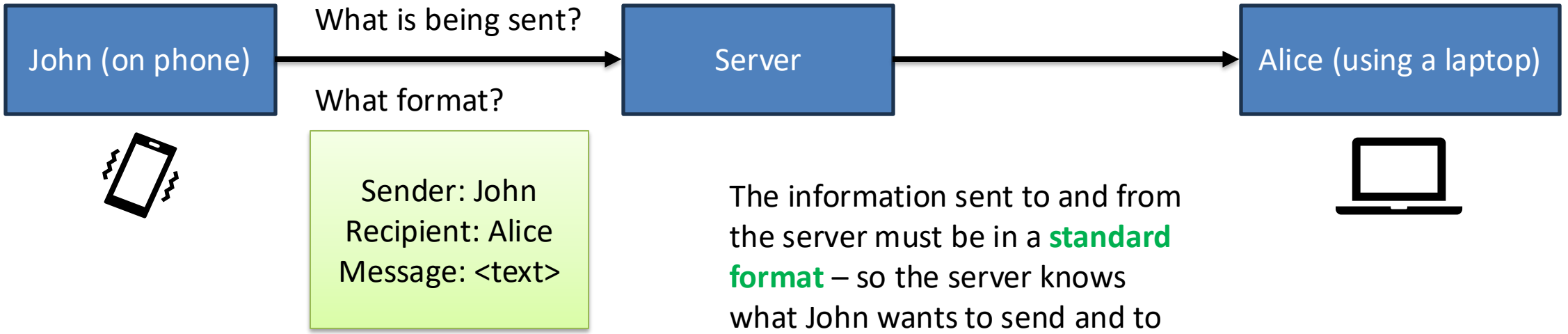
USING WEB SERVICES

Data on the Web



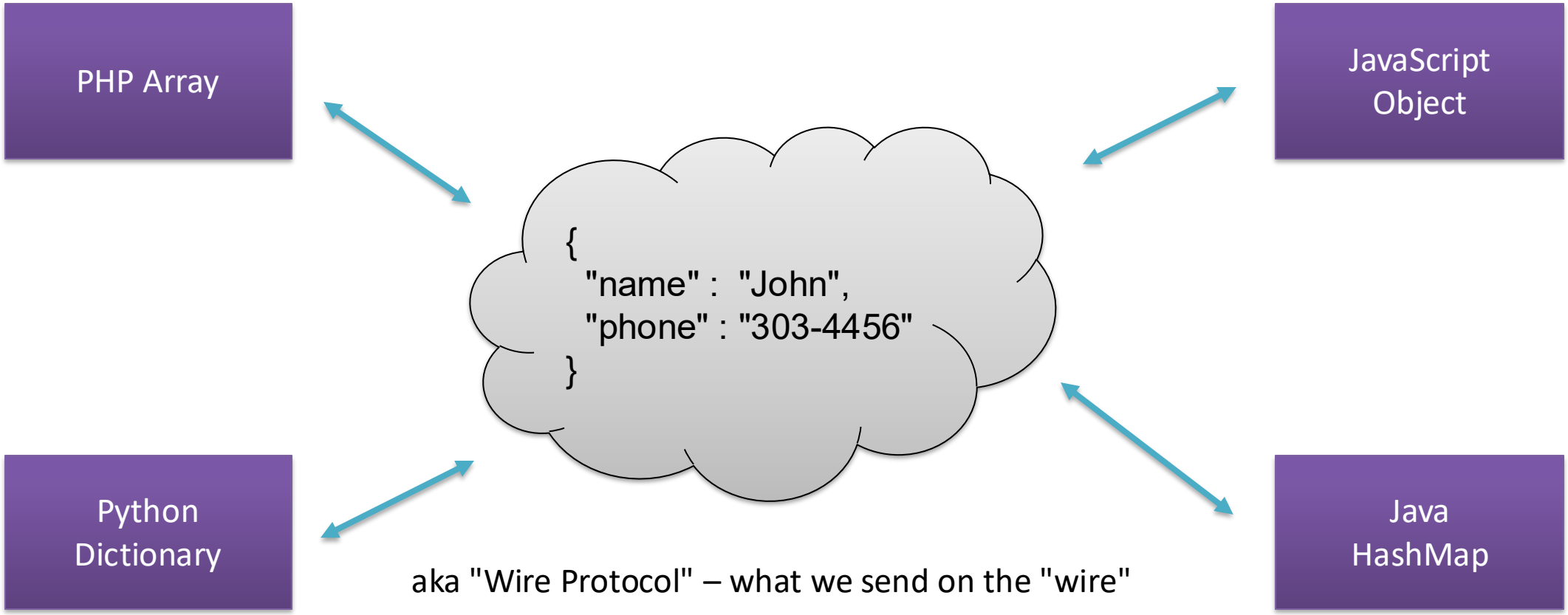
- With the HTTP Request/Response well understood and well supported, there was a natural move toward exchanging data between programs using these protocols
- We needed to come up with an agreed upon way to represent data going between applications and across networks
- There are two commonly used formats: XML and JSON

Example: Texting Application on the Phone



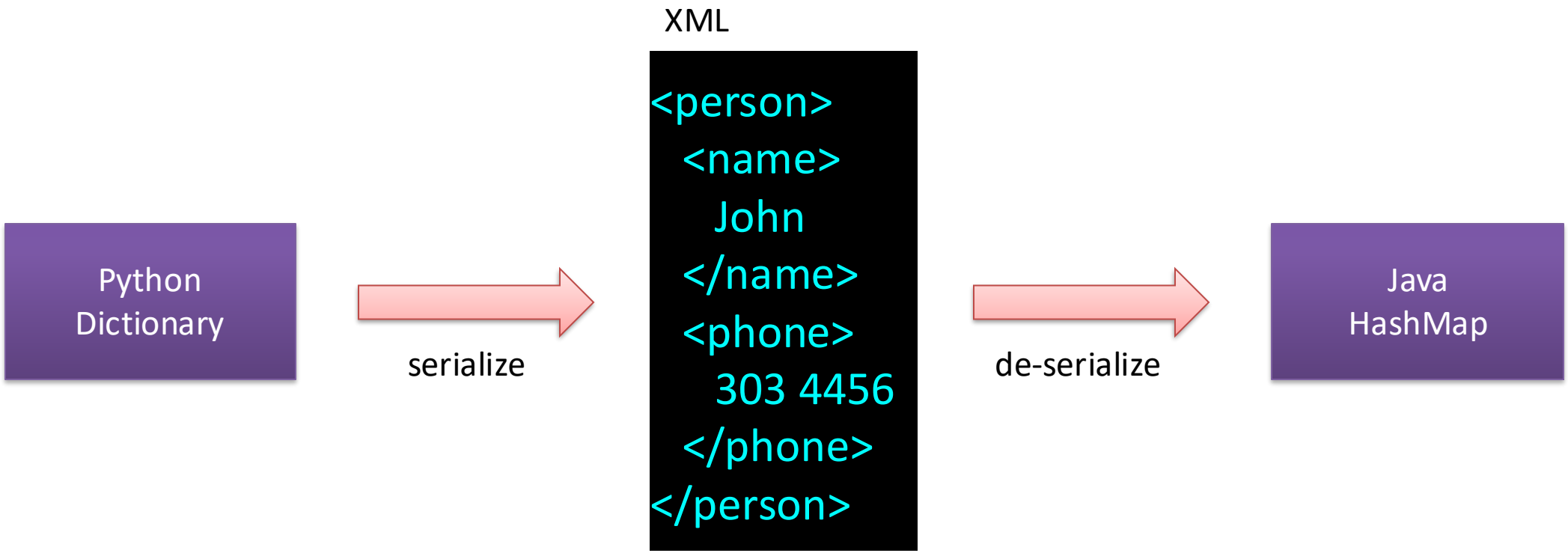
The information sent to and from the server must be in a **standard format** – so the server knows what John wants to send and to whom, and Alice will know who it came from and what it says

Sending Data across the "Net"

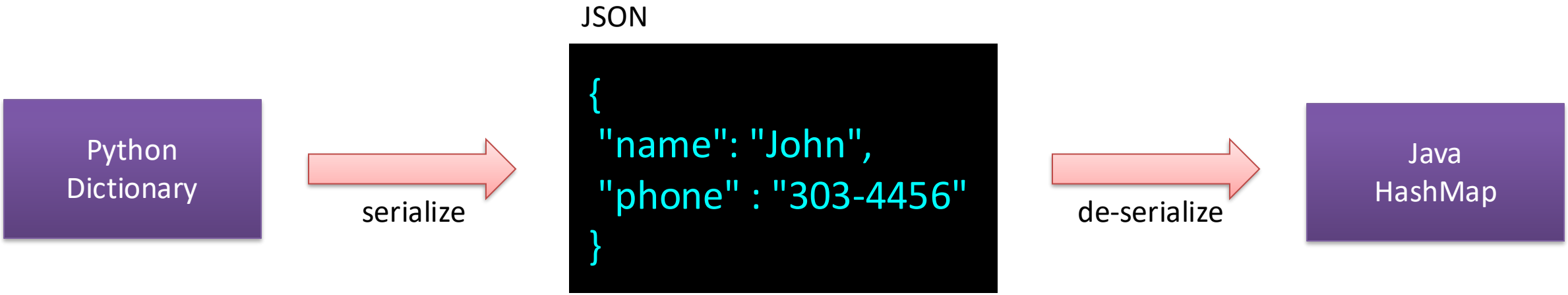


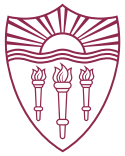
aka "Wire Protocol" – what we send on the "wire"

Agreeing on a "Wire Format"



Agreeing on a "Wire Format"





eXtensible Markup Language - https://en.wikipedia.org/wiki/XML

XML

XML

```
<person>
   <name> John </name>
   <phone> 303-4456 </phone>
</person>
```

- Primary purpose is to help information systems **share structured data**
- It started as a simplified subset of the Standard Generalized Markup Language (**SGML**), and is designed to be relatively human-legible

Problems with HTML

```
<html>
    <head>
        <title>My title</title>
    </head>
    <body>
        <h1>A heading</h1>
        <a href="https://www.usc.edu"> Link text</a>
     </body>
</html>
```

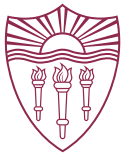
Problems with HTML:

- HTML is **intended for presentation** of information as Web pages.
- HTML contains a **fixed set of markup tags**.

This design is not appropriate for data:

- Tags don't convey meaning of the data inside the tags.
- Tags are not extensible.

XML (eXtensible Markup Language)



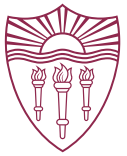
- HTML:

```
<h2>Nonmonotonic Reasoning: Context-Dependent Reasoning</h2>
<i>by <b>V. Marek</b> and <b>M. Truszczynski</b>
</i><br>
Springer 1993<br>
ISBN 0387976892
```

- XML:

```
<book>
    <title>Nonmonotonic Reasoning: Context-Dependent Reasoning</title>
    <author>V. Marek</author>
    <author>M. Truszczynski</author>
    <publisher>Springer</publisher>
    <year>1993</year>
    <ISBN>0387976892</ISBN>
</book>
```

Design of XML



- Tags can be used to indicate the meaning of data/information
- There is no fixed set of markup tags - new tags can be defined
- Underlying data model is a **tree structure**
- XML provides a common exchange format

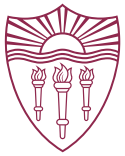
HTML vs XML

HTML	XML
Hyptert**t**ext **M**arkup **L**anguage	e**X**tensible **M**arkup **L**anguage
For web presentation of data	For transfer of data
Predefined tags	User-defined tags
Case insensitive	Case sensitive
Lenient regarding format errors	Strict regarding format errors
Since 1991	Since 1998
Latest Version: HTML 5.3	Latest version: XML 1.1

But HTML and XML have the basic syntactic structure in common:

<tag attribute="value">text content</tag>

XML "Elements" (or "Nodes")

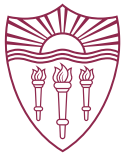


- Simple Element
- Complex Element

```
<people>
  <person>
    <name>John</name>
    <phone>303 4456</phone>
  </person>
  <person>
    <name>Noah</name>
    <phone>622 7421</phone>
  </person>
</people>
```

In XML, only one root element is allowed (here: <people>),
but child elements (here: <person>) can appear repeatedly within it.

XML Basics

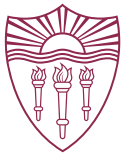


- Start Tag
- End Tag
- Text Content
- Attribute
- Self Closing Tag

Line ends do not matter. White space is generally discarded on text elements. We indent only to be readable.

```
<person>
  <name>John</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>
```

XML Terminology



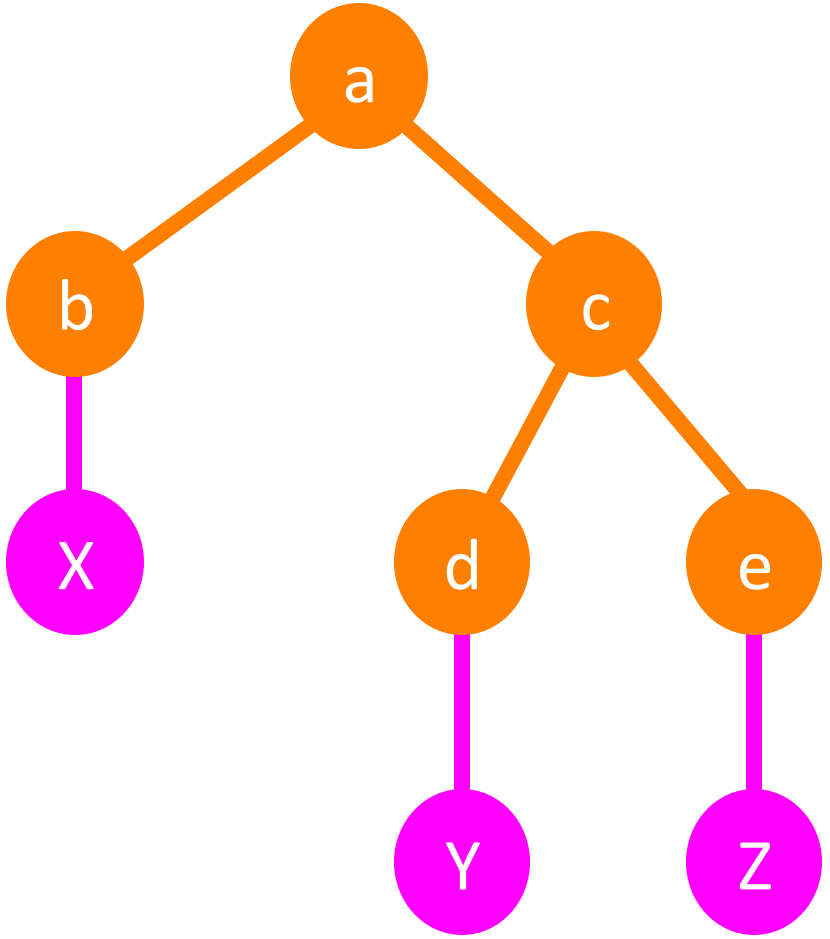
- **Tags** mark the beginning and end of elements in XML.
  - Most elements have both an opening tag and a closing tag, although some elements may use a self-closing tag instead.
- **Attributes** - Keyword/value pairs on the opening tag of XML
- **Serialize / De-Serialize** - Convert data in one program into a common format that can be stored and/or transmitted between systems in a programming language-independent manner

http://en.wikipedia.org/wiki/Serialization

XML as a Tree

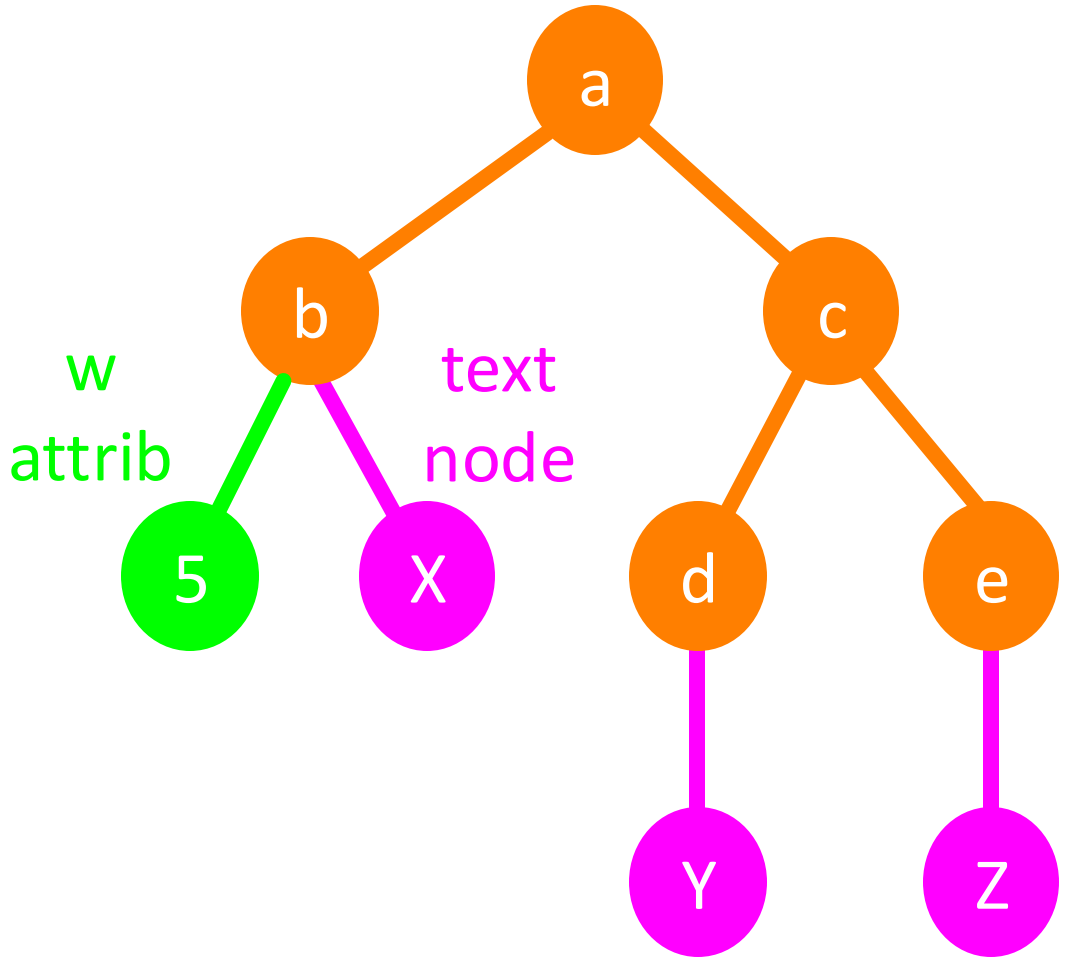
```
<a>
  <b>X</b>
  <c>
    <d>Y</d>
    <e>Z</e>
  </c>
</a>
```

Elements Text



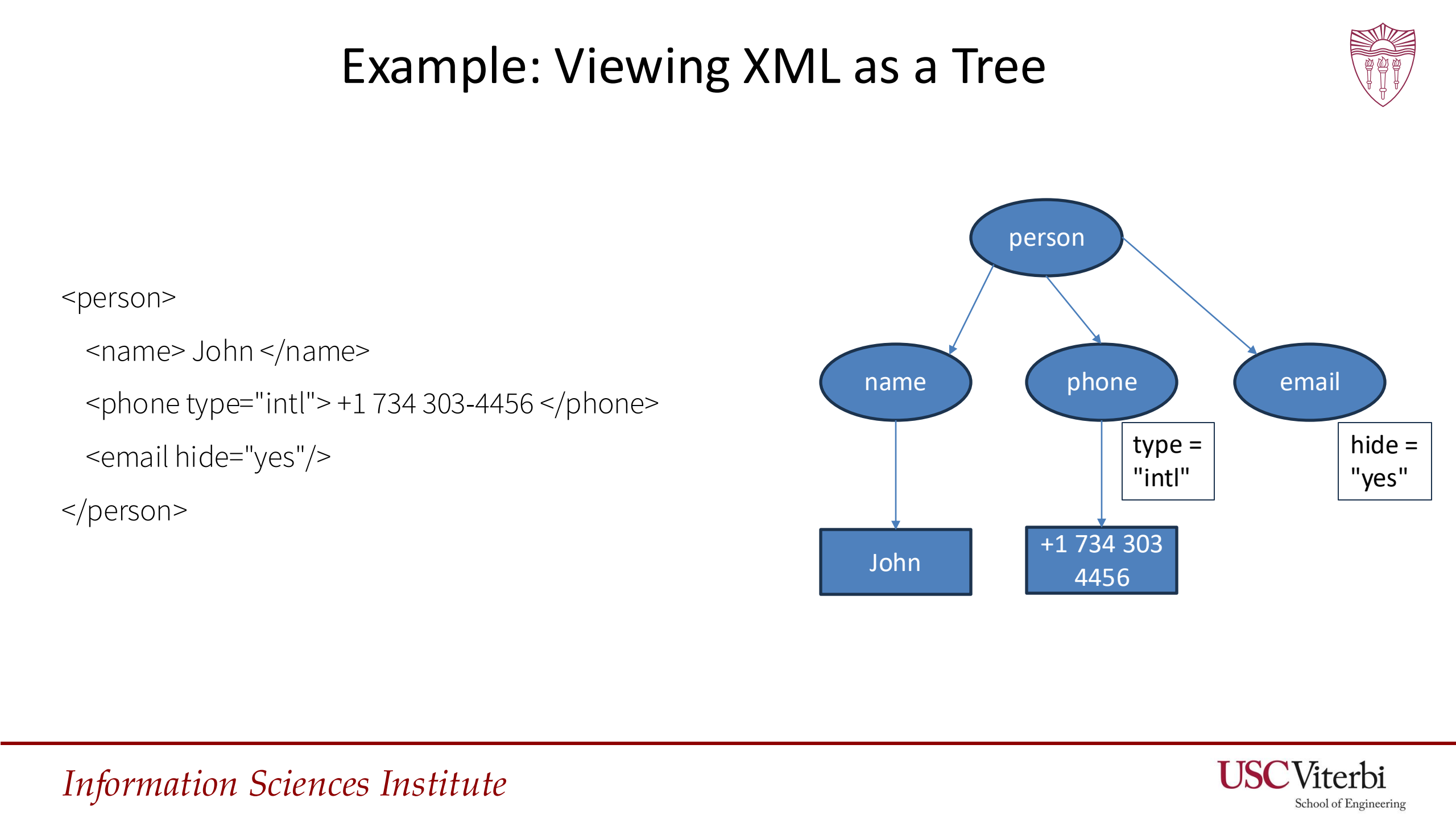
XML Text and Attributes

```
<a>
 <b w="5">X</b>
 <c>
  <d>Y</d>
  <e>Z</e>
 </c>
</a>
```



Elements Text

Example: Viewing XML as a Tree

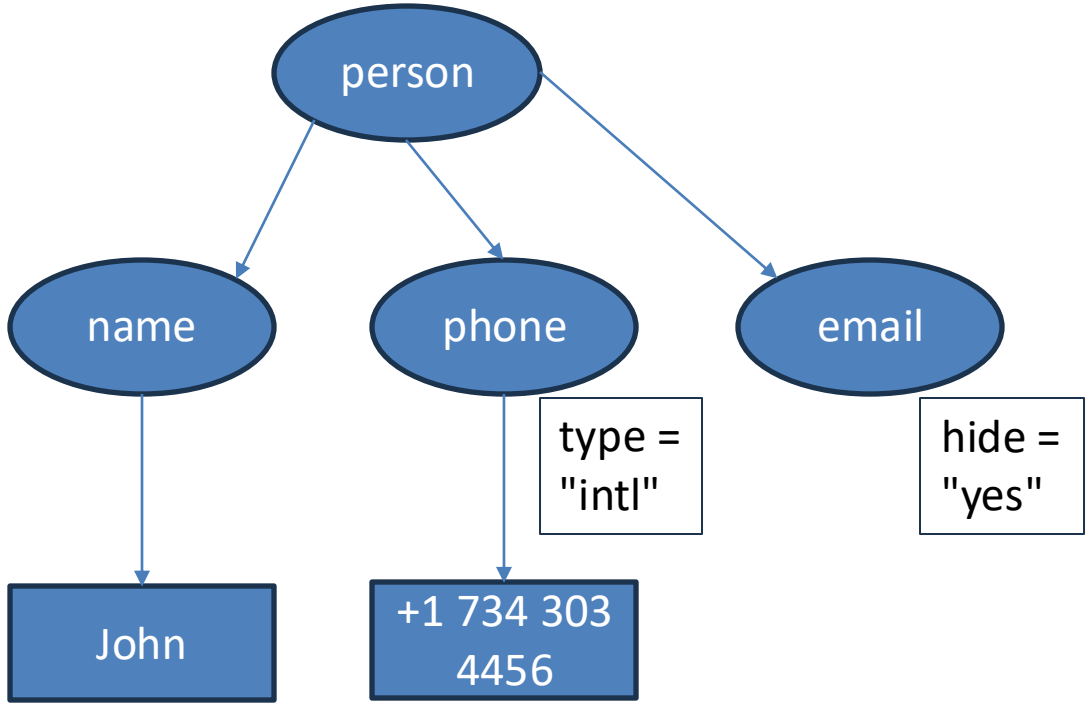


```
<person>
   <name> John </name>
   <phone type="intl"> +1 734 303-4456 </phone>
   <email hide="yes"/>
</person>
```

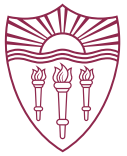
Example: Viewing XML as a “Path”

person/name → John

person/phone → +1 734 303-4456



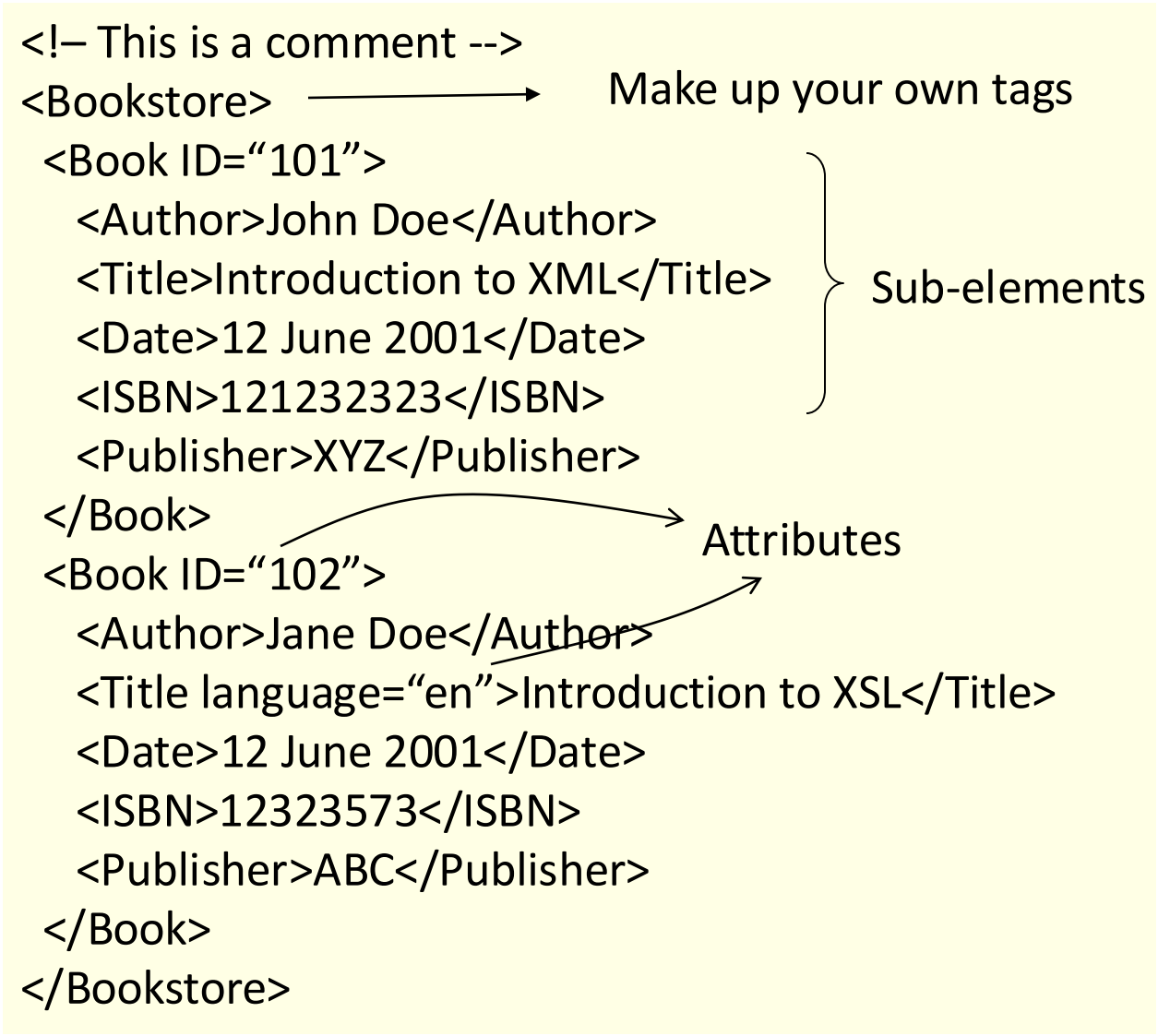
There's also XML Meta data



- **XML Declaration** – <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  Declares version, encoding, and standalone status of the document.
- **Comments** – <!-- This section lists all authors -->
  Human-readable notes ignored by XML parsers.
- **Document Type Declaration (DTD)** – <!DOCTYPE note SYSTEM "note.dtd">
  Points to or defines a DTD that specifies allowed structure.

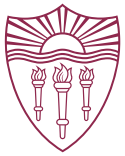
- **Namespaces** – xmlns / xmlns:prefix
  Identify the vocabulary of elements and prevent name conflicts.
- **Schema References** – xsi:schemaLocation / xsi:noNamespaceSchemaLocation
  Link the document to an XML Schema (XSD) for validation.

Another XML Example



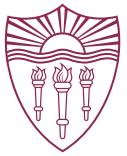
- XML by itself is just a hierarchically structured text
- Choice of element vs attribute often matter of taste, but attributes cannot be nested

Namespaces



- A Namespace indicates the scope within which an element (name) is valid
- Why do we need Namespaces ?
  - If elements were defined within a global scope, it becomes a problem when combining elements from multiple documents
  - Modularity: If a markup vocabulary exists which is well understood and for which there is useful software available, it is better to reuse it
- Namespaces in XML:
  - An XML namespace is a collection of names, identified by a URI reference.
  - A URI reference (Uniform Resource Identifier reference) is a string that identifies or points to a resource — such as a document, file, or concept — on the web or within a namespace system.
  - Names from XML namespaces may appear as qualified names, which contain a single colon, separating the name into a prefix and a local part. The prefix, which is mapped to a URI reference, selects a namespace.

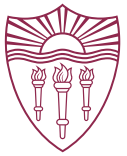
Namespaces: An example



```
<x xmlns:edi="http://ecommerce.example.org/schema">
   <edi:price units="Euro">32.18</edi:price>
</x>
```

- <x> is the root element of this XML document.
- It contains a **namespace declaration**:
  xmlns:edi="http://ecommerce.example.org/schema"
- xmlns means **“XML namespace.”**
- edi is the prefix — a shorthand label we may use for elements in this namespace.
- "http://ecommerce.example.org/schema" is a **URI reference** that **identifies the namespace**.
- This URI does not need to be an actual web address —
  it’s just a unique identifier for the namespace vocabulary.

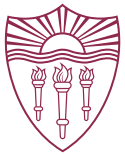
Namespaces: An example



```
<x xmlns:edi="http://ecommerce.example.org/schema">
   <edi:price units="Euro">32.18</edi:price>
</x>
```

Inside <x> we have:
<edi:price units="Euro">32.18</edi:price>

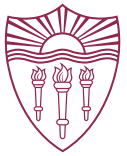
Component	Meaning
edi:	Prefix that tells XML this element belongs to the namespace bound to edi.
price	The **local name** of the element. Together with the prefix, it forms the **qualified name** edi:price.
units="Euro"	An **attribute** of the element, describing the currency.
32.18	The **text content** of the element — in this case, the price value.



Describing a "contract" as to what is acceptable XML - http://en.wikipedia.org/wiki/Xml_schema

XML SCHEMA DEFINITION

XML Schema



- Description of the **legal format** of an XML document
- Expressed in terms of constraints on the structure and content of documents
- Often used to specify a "**contract**" between systems – "My system will only accept XML that conforms to this particular Schema."
- If a particular piece of XML meets the specification of the Schema - it is said to "**validate**"

http://en.wikipedia.org/wiki/Xml_schema
http://www.w3.org/XML/Schema

XML Schema Example

```
<person>
  <lastname>Severance</lastname>
  <age>22</age>
  <dateborn>2001-04-17</dateborn>
</person>
```

XML Document

XML Validation

```
<xs:complexType name=”person”>
 <xs:sequence>
  <xs:element name="lastname" type="xs:string"/>
  <xs:element name="age" type="xs:integer"/>
  <xs:element name="dateborn" type="xs:date"/>
 </xs:sequence>
</xs:complexType>
```

XML Schema Definition (XSD)

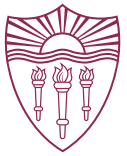
XSD Structure

```
<person>
  <lastname>Severance</lastname>
  <age>17</age>
  <dateborn>2001-04-17</dateborn>
</person>

<xs:complexType name=”person”>
 <xs:sequence>
  <xs:element name="lastname" type="xs:string"/>
  <xs:element name="age" type="xs:integer"/>
  <xs:element name="dateborn" type="xs:date"/>
 </xs:sequence>
</xs:complexType>
```

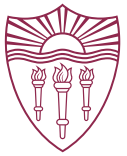
- xs:element
- xs:sequence
- xs:complexType

Structural building blocks of XSD



Tag	Purpose	Typical usage
`xs:element`	Defines an XML element (tag)	The *most important*. Every XML node comes from here.
`xs:complexType`	Defines an element with **children** and/or **attributes**	Use when an element contains nested tags.
`xs:simpleType`	Defines an element with **only text content** (no children)	Use when restricting or enumerating literal values.
`xs:attribute`	Defines an **attribute** (a property inside a tag)	For things like `id="bk1"` or `units="Euro"`.
`xs:sequence`	Children appear **in order**	`<title>` must come before `<price>`.

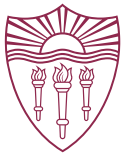
XSD Constraints



```
<xs:element name="person">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="full_name" type="xs:string"
      minOccurs="1" maxOccurs="1" />
    <xs:element name="child_name"
type="xs:string"
       minOccurs="0" maxOccurs="10" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

```
<person>
 <full_name>Tove Refsnes</full_name>
 <child_name>Hege</child_name>
 <child_name>Stale</child_name>
 <child_name>Jim</child_name>
 <child_name>Borge</child_name>
</person>
```

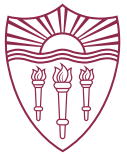
Parsing XML: BeautifulSoup vs ElementTree



BeautifulSoup	xml.etree.ElementTree
from bs4 import BeautifulSoup as bs	import xml.etree.ElementTree as et
tree = bs(data, 'xml')	tree = et.fromstring(data)
person = tree.find('person')	person = tree.find('person')
people = tree.findAll('person')	people = tree.findAll('person')
id = person.attrs.get('id')	id = person.get('id')
person.string	person.text
person.attrs # gives an element's attributes	tree.attrib

```
<friends>
  <person id="0123">Charlie Brown</person>
  <person id="4567">Lucy van Pelt</person>
</friends>
```

Parsing XML: ElementTree



```
data = """
<library>
  <book title="The Hobbit" author="J.R.R. Tolkien" year="1937">
    <summary>Bilbo Baggins goes on an unexpected journey.</summary>
  </book>
  <book title="Dune" author="Frank Herbert" year="1965">
    <summary>Paul Atreides becomes the ruler of Arrakis.</summary>
  </book>
</library>
"""
```

```
import xml.etree.ElementTree as ET
tree = ET.fromstring(data)
```

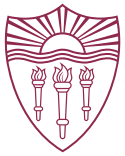
```
print(tree.tag)
```

```
library
```

```
for book in tree:
    print(book.tag, book.attrib)
```

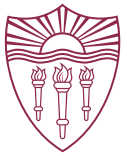
```
book {'title': 'The Hobbit', 'author': 'J.R.R. Tolkien', 'year': '1937'}
book {'title': 'Dune', 'author': 'Frank Herbert', 'year': '1965'}
```

ElementTree: Parsing XML



- fromstring converts the string representation of the XML into a "tree" of XML nodes.
- When the XML is in tree form, we have a series of methods we can call to extract portions of data from the XML.
- The find function searches through the XML tree and retrieves a node that matches the specified tag.
- Each node can have some text, some attributes (like *hide*, which we saw earlier), and some “child” nodes.
- Each node can be the top of another tree of nodes.

Parsing XML: BeautifulSoup



```
from bs4 import BeautifulSoup
data = """
<library>
  <book title="The Hobbit" author="J.R.R. Tolkien" year="1937">
    <summary>Bilbo Baggins goes on an unexpected journey.</summary>
  </book>
  <book title="Dune" author="Frank Herbert" year="1965">
    <summary>Paul Atreides becomes the ruler of Arrakis.</summary>
  </book>
</library>
"""
soup = BeautifulSoup(data, "xml")   # use the XML parser, not "html.parser"
```

```
books = soup.find_all("book")
print(len(books))
```

```
2
```

```
for book in books:
    title = book["title"]
    author = book.get("author")
    summary = book.summary.text.strip()
    print(f"{title} by {author}: {summary}")
```