5. JSON, XML

5.1. JSON

5.2. XML

5.1. JSON

■ What is JSON?

- ○ JSON stands for JavaScript Object Notation
- ○ Designed for human-readable data interchange
- ○ Extended from the JavaScript scripting language
- ○ JSON Internet Media type is **application/json**, extention is **.json**

■ Why use JSON?

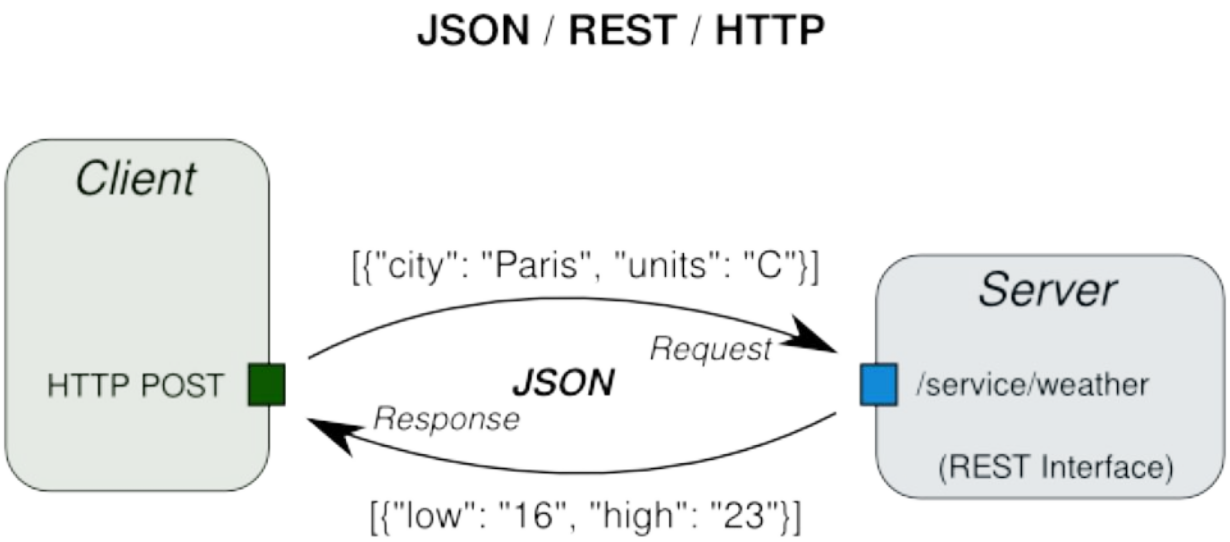
○ Used for serializing and transmitting structured data over network connection

■ Characteristics

○ Easy to read and write

○ Programming Language independent

○ A lightweight text-based interchange format



■ 예제

```
{
   “Make”   : “Volkswagen”,
   “Year”   : 2003,
   “Model” : {
                “Base” : “Golf”,
                “Trim” : “GL”
             },
   “Colors” : [“White”, “Pearl”, “Rust”],
   “PurchaseDate” : “2006-10-05T00:00:00.000Z”
}
```

○ Easy processing

Javascript:

```
var car = { "Make" : "Volkswagen" };
console.log(car.Make);
// Output: Volkswagen

car.Year = 2003;
console.log(car);
// Output: { "Make" : "Volkswagen", "Year" : 2003" }
```

○ Storage size

XML: 225 Characters

```
<Car>
  <Make>Volkswagen</Make>
  <Year>2003</Year>
  <Model>
    <Base>Golf</Base>
    <Trim>GL</Trim>
  </Model>
  <Colors>
    <Color>White</Color>
    <Color>Pearl</Color>
    <Color>Rust</Color>
  </Colors>
  <PurchaseDate>
    2006-10-05 00:00:00.000
  </PurcaseDate>
</Car>
```

JSON: 145 Characters

```
{
  “Make” : “Volkswagen”,
  “Year” : 2003,
  “Model” : {
              “Base” : “Golf”,
              “Trim” : “GL”
            },
  “Colors” :
      [“White”, “Pearl”, Rust”],
  “PurchaseDate” :
      “2006-10-05T00:00:00.000Z”
}
```

○ JSON API

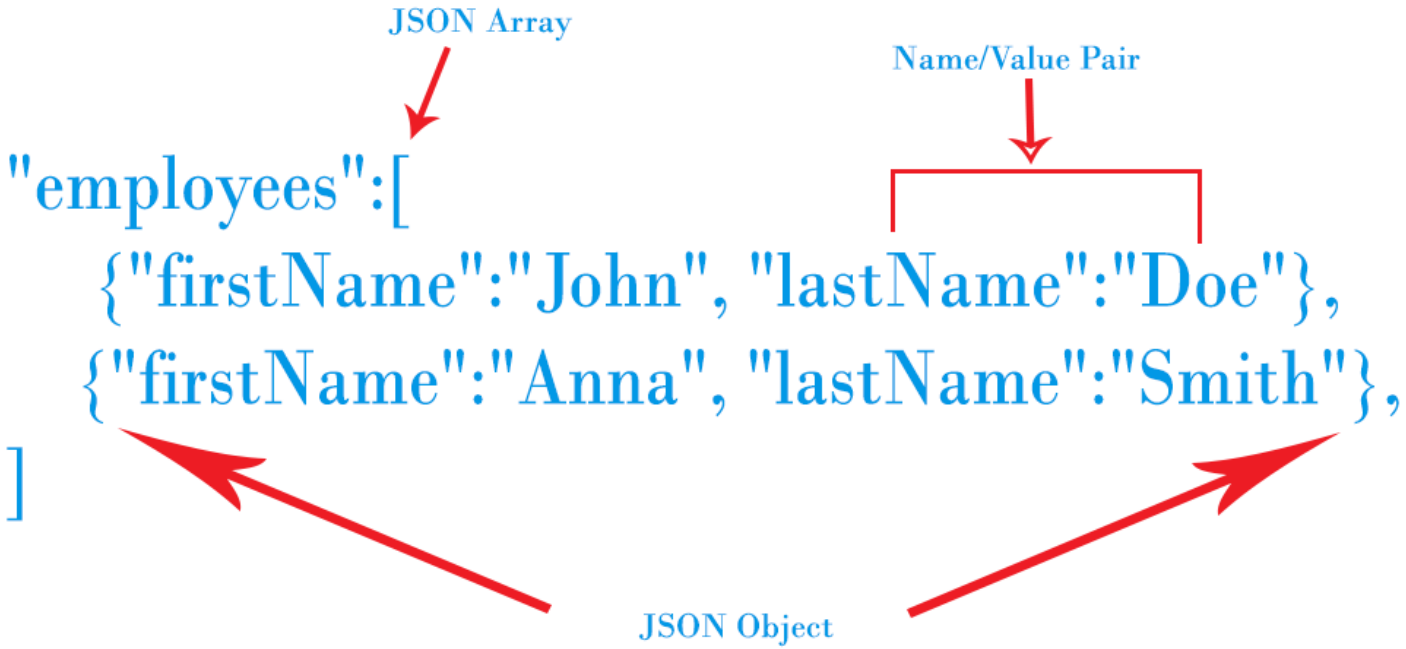
{ json:api }

기상청에서 제공하는 Open-API 목록을 조회하고 활용 신청할 수 있도록 링크를 제공합니다.

전체 27건 선택 검색

번호	서비스 명	유형	자료포멧	등록일	조회수	상세보기
27	낙뢰정보 낙뢰정보조회서비스	REST	XML	2018-03-20	228	
26	태풍정보 태풍정보조회서비스	REST	JSON	2018-03-20	156	
25	동네예보통보문 동네예보통보문조회서비스	REST	JSON	2018-03-20	684	
24	항공기상전문 항공기상전문서비스	REST	XML	2018-03-20	82	
23	세계 주요공항 항공기상전문 세계 주요공항 항공기상전문 서비스	REST	XML	2018-03-20	23	

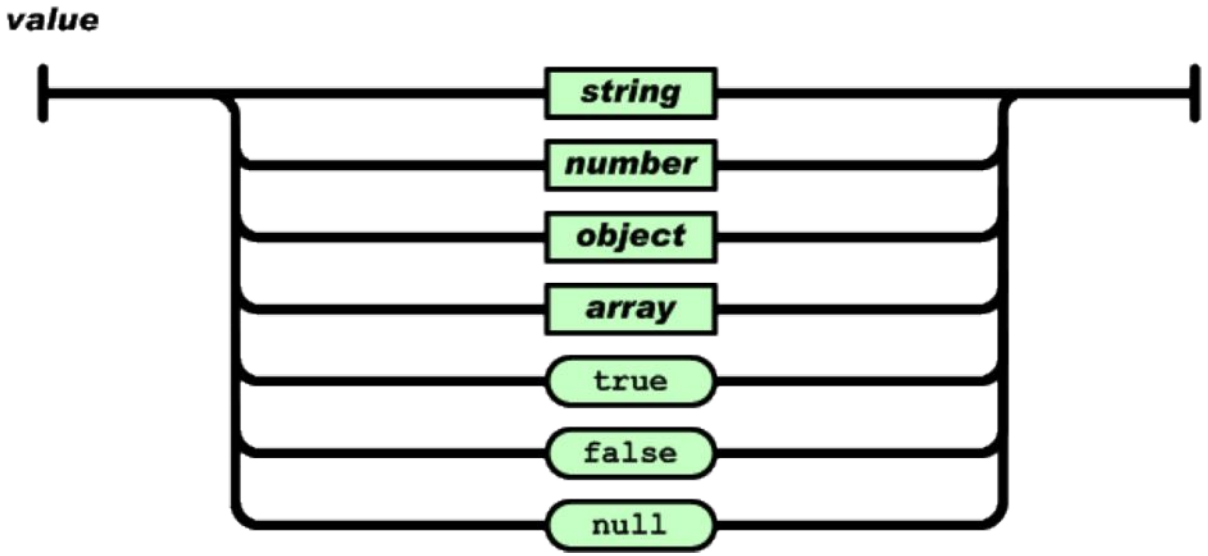
■ JSON Concept

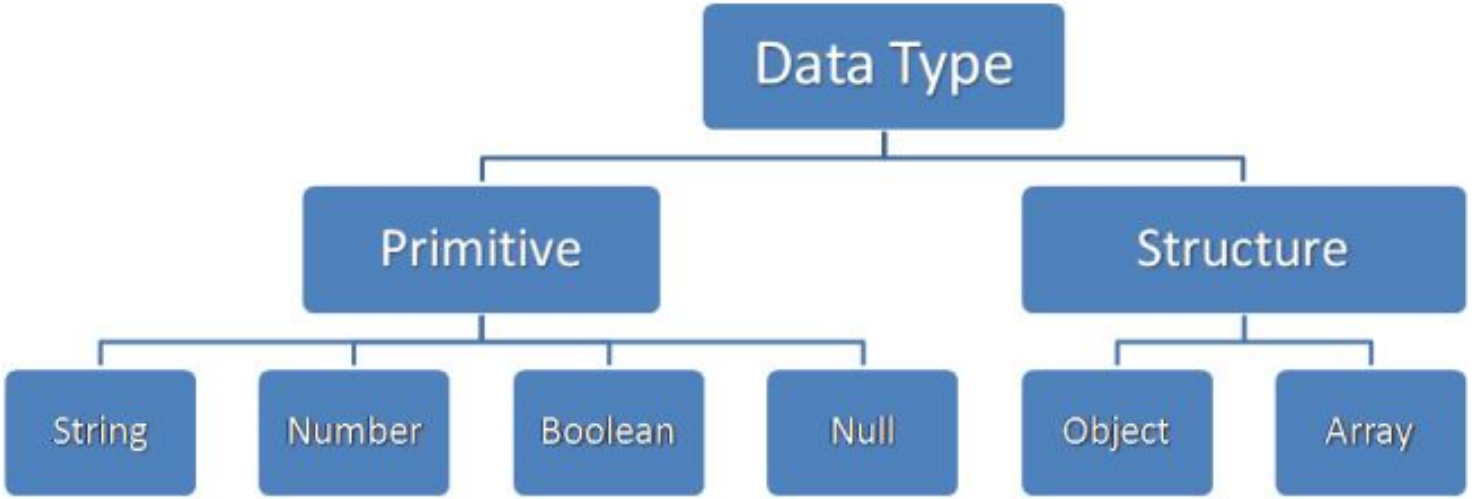


○ JSON Object
- ➢ name, value pairs

○ JSON Array
- ➢ Ordered collections

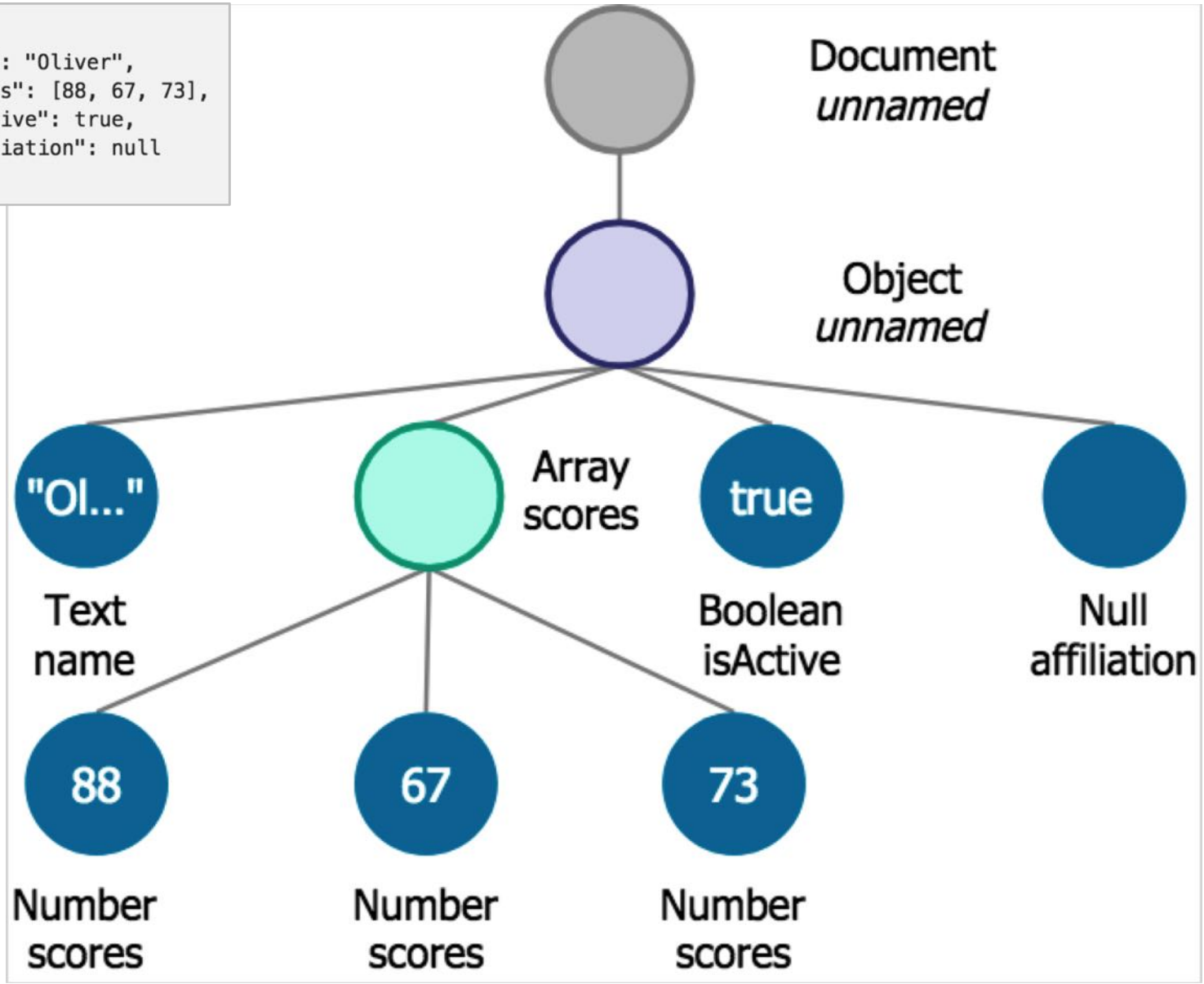
■ JSON Data Type





■ JSON Hierarchy

```
{
 "name": "Oliver",
 "scores": [88, 67, 73],
 "isActive": true,
 "affiliation": null
}
```



■ 예제

○ JSON data types

- String
- Number
- Boolean
- Null
- Object
- Array

```
<script>
  var who = { "name" : "Kim" };
  var age = { "age" : 30 };
  var car = { "car" : true };
  var middle = { "middle name" : null };

  var personObj = { "name" : "kim", "age" : 30, "car" : true, "middle name" : null };

  var personList = { "list" : [ "Kim", "Lee", "Park" ] };
</script>
```

○ Accessing object

```
<script>
  var personObj = { "name" : "kim", "age" : 30, "car" : true, "middle name" : null };

  var name1 = personObj.name;
  var name2 = personObj["name"];

  console.log(name1, name2);
</script>
```

○ Looping an object

```
<script>
  var personObj = { "name" : "kim", "age" : 30, "car" : true, "middle name" : null };

  for (x in personObj) {
      console.log(x, personObj[x]);
  }
</script>
```

○ Nested objects

```
<script>
  var personObj = { "name" : "kim",
                    "age" : 30,
                    "car" : {
                        "car1" : "Hyundai",
                        "car2" : "Kia",
                        "car3" : "Chevrolet"
                    }
                  };

  console.log(personObj.car.car1);
</script>
```

○ Modify values

```
<script>
  var personObj = { "name" : "kim",
                    "age" : 30,
                    "car" : {
                        "car1" : "Hyundai",
                        "car2" : "Kia",
                        "car3" : "Chevrolet"
                    }
                  };

  personObj.car.car1 = "Mercedes";

  console.log(personObj.car.car1);
</script>
```

○ Delete object properties

```
<script>
  var personObj = { "name" : "kim",
                    "age" : 30,
                    "car" : {
                        "car1" : "Hyundai",
                        "car2" : "Kia",
                        "car3" : "Chevrolet"
                    }
                  };

  delete personObj.car.car3;

  console.log(personObj.car);
</script>
```

○ Accessing array values

```
<script>
  var personObj = { "name" : "kim", "age" : 30,
                    "cars" : [ "Hyundai", "Kia", "Chevrolet" ] };

  console.log(personObj.cars[0]);
</script>
```

○ Looping through an array

```
<script>
  var personObj = { "name" : "kim", "age" : 30,
                    "cars" : [ "Hyundai", "Kia", "Chevrolet" ] };

  for (x in personObj.cars) {
      console.log(personObj.cars[x]);
  }

  for (i=0; i<personObj.cars.length; i++) {
      console.log(personObj.cars[i]);
  }
</script>
```

○ Nested arrays

```
<script>
  var personObj = { "name" : "kim", "age" : 30,
                    "cars" : [
                      { "name" : "Hyundai", "models" : [ "Avante", "Sonata", "grandeur" ] },
                      { "name" : "Kia", "models" : [ "K3", "K5", "K7"] },
                      { "name" : "Chevrolet", "models" : [ "Cruze", "Malribu", "Impala"] }
                    ] };

  for (i in personObj.cars) {
      console.log(personObj.cars[i].name);

    for (j in personObj.cars[i].models) {
        console.log(personObj.cars[i].models[j]);
    }
  }
</script>
```

○ Delete array items

➢ Shift, Pop, Splice

```
delete personObj.cars[2].models[0];
console.log(personObj.cars);

delete personObj.cars[2];

console.log(personObj.cars);
```

○ Stringify

- a common use of JSON is to exchange data to/from a web server
- when sending data to a web server, the data has to be a string

```
JSON.stringify(value[, replacer[, space]])
```

value – The JSON object to convert to a JSON string.

replacer – A function that alters the behavior of the stringification process. If this value is null or not provided, all properties of the object are included in the resulting JSON string.

space – A String or Number object that's used to insert white space into the output JSON string for readability purposes. If this is a Number, it indicates the number of space characters to use as white space.

- Javascript object

```
<script>
  var obj = { name : "Kim", "age" : 30, "car" : true };

  console.log(obj);
  console.log(JSON.stringify(obj));

  objStr = JSON.stringify(obj);

  console.log(obj.name);
  console.log(objStr.name);
</script>
```

➢ Javascript array

```
<script>
  var arr = [ "Kim", 30, true ];

  console.log(arr);
  console.log(JSON.stringify(arr));

  arrStr = JSON.stringify(arr);

  console.log(arr[0]);
  console.log(arrStr[0]);
</script>
```

➢ replacer, space

```
<script>
  var obj = { name : "Kim", "age" : 30, "car" : true };

  console.log(obj);
  console.log(JSON.stringify(obj));

  objStr = JSON.stringify(obj, ["name"]);

  console.log(obj);
  console.log(objStr);

  objStr = JSON.stringify(obj, function(k,v){if(v==="Kim")return;else return v;});

  console.log(obj);
  console.log(objStr);

  objStr = JSON.stringify(obj, null, 10/* ' ', '\t' */);

  console.log(obj);
  console.log(objStr);
</script>
```

○ Parse

- a common use of JSON is to exchange data to/from a web server
- When receiving data from a web server, the data is always a string

```
JSON.parse(text[, reviver])
```

text – The string to parse as JSON object.

reviver – A function that prescribes how the value originally produced by parsing is transformed, before being returned.

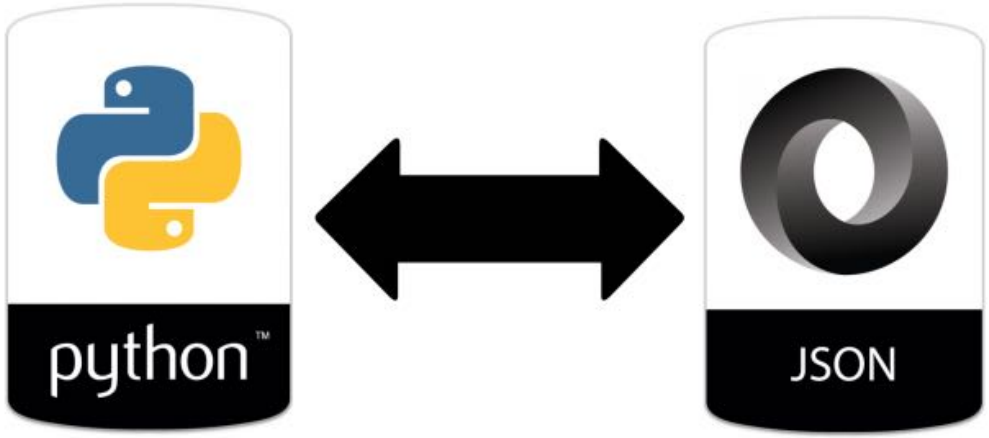
- parse

```
<script>
  var str = '{ "name" : "Kim", "age" : 30, "car" : ["Ray", "Spark"] }';

  console.log(str);
  console.log(str.name);

  obj = JSON.parse(str);

  console.log(obj);
  console.log(obj.name);
</script>
```

○ Reviver

```
<script>
  var str = '{ "name" : "Kim", "age" : 30, "car" : ["Ray", "Spark"] }';

  console.log(str);
  console.log(str.name);

  obj = JSON.parse(str, function(k,v) {
                          if (typeof(v) == "number")
                              return v.toString();
                          else
                              return v; } );

  console.log(obj);
  console.log(obj.age);
</script>
```

■ JSON with Python



○ JSON library

```
import json
```

○ Encoding

- ➢ dump, dumps

○ Decoding

- ➢ load, loads

○ Data types

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

○ dump / dumps

```
json.dump(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True,
cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)
```

```
json.dumps(obj, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True,
cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)
```

Serialize *obj* to a JSON formatted `str` using this conversion table. The arguments have the same meaning as in `dump()`.

```
name = ("Kim", "Lee", "Park")
age = [30, 28, 31]
person = {"name":"Kim", "age":30, "car":False}
```

```
nameStr = json.dumps(name)
ageStr = json.dumps(age)
personStr = json.dumps(person, indent="  ")
```

```
kname = ("김", "이", "박")
knameStr = json.dumps(kname)
print(knameStr)

knameStr = json.dumps(kname, ensure_ascii=False)
print(knameStr)
```

○ load / loads

```
json.load(fp, *, cls=None, object_hook=None, parse_float=None, parse_int=None,
parse_constant=None, object_pairs_hook=None, **kw)
```

```
json.loads(s, *, encoding=None, cls=None, object_hook=None, parse_float=None,
parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)
```

Deserialize *s* (a `str`, `bytes` or `bytearray` instance containing a JSON document) to a Python object using this conversion table.

```
personObj = json.loads(personStr)
person == personObj
```

○ With a file

```
person["car"] = ["레이", "모닝"]

with open("person.json", "w") as f:
    json.dump(person, f)
```

```
with open("person.json", "r") as f:
    personObj = json.load(f)

personObj
```

○ 예제

```
import json
import urllib.request

url = "http://ip.jsontest.com"

obj = {"name":"김이박", "age":30}
objStr = json.dumps(obj)
objByte = objStr.encode("utf-8")

req = urllib.request.Request(url, data=objByte, headers={'content-type': 'application/json'})
res = urllib.request.urlopen(req)

resByte = res.read()
resStr = resByte.decode("utf-8")
resObj = json.loads(resStr)

print(resByte, type(resByte))
print(resStr, type(resStr))
print(resObj, type(resObj))
```

○ 전국 대기오염도 현황 Open API

- 서비스
  - https://www.data.go.kr/subMain.jsp#/L3B1YnIvcG90L215cC9Jcm9zTXlQYWdlL29wZW5EZXZHdWlkZVBhZ2UkQF4wMTIkQF5wdWJsaWNEYXRhRGV0YWlsUGs9dWRkaTo3MDkxMTBlNy1kN2IxLTQ0MjEtOTBiYS04NGE2OWY5ODBjYWJfMjAxNjA4MDgxMTE0JEBebWFpbkZsYWc9dHJ1ZQ==
- URL
  - http://openapi.airkorea.or.kr/openapi/services/rest/ArpltnInforInqireSvc/getMsrstnAcctoRltmMesureDnsty
- Parameters
  - stationName
  - dateTerm = *daily* | *month* | *3month*
  - pageNo
  - numOfRows
  - ver = *1.1* | *1.2* | *1.3* | *1.4*
  - _returnType = *json*

○ 전국 대기오염도 현황 Open API 활용

```
url = "http://openapi.airkorea.or.kr/openapi/services/rest/ArpltnInforInqireSvc/getMsrstnAcctoRltmMesureDnsty"

params = {
    "serviceKey":"k6C4a%2FFJDFxsVHTVxI2p0%2F1ZYidISwqO6LPY9LSoqFoAB6AxIA9vn9e1uB8j48P5h5xs9hO4VEpS%2BpJbRgiXJQ%3D%3D",
    "numOfRows":10,
    "pageSize":10,
    "pageNo":1,
    "startPage":1,
    "stationName":"성북구",
    "dataTerm":"DAILY",
    "ver":"1.3",
    "_returnType":"JSON"
}

params["serviceKey"] = urllib.parse.unquote(param["serviceKey"])
params = urllib.parse.urlencode(params)
params = params.encode("utf-8")

req = urllib.request.Request(url, data=params)
res = urllib.request.urlopen(req)

resStr = res.read()

resStr = resStr.decode("utf-8")
resObj = json.loads(resStr)

resJSON = json.dumps(resObj, indent="   ")
print(resJSON)
```

5.2. XML

■ What is XML?

- ○ XML stands for eXtensible Markup Language
- ○ Designed to store and transport data
- ○ Designed to be both human- and machine-readable
- ○ XML is a W3C Recommendation

■ Why use XML?

○ Universally accepted standard way of structuring data

○ Provides a well-defined structure for communication

○ Software- and hardware-independent tool for storing and transporting data

■ Characteristics

○ Text-based (Unicode)

- ➢ more readable, easier to document, easier to debug

○ No predefined tags

- ➢ the author must define both the tags and the document structure

○ Extensible

- ➢ supporting rich structure, like objects or hierarchies or relationships
- ➢ most XML applications will work as expected even if new data is added (or removed).

○ Validity

- ➢ supporting validation and well-formed properties

■ Well-formed Documents

○ An XML document with correct syntax is called **Well Formed**

- ➢ XML documents must have a root element
- ➢ XML elements must have a closing tag
- ➢ XML tags are case sensitive
- ➢ XML elements must be properly nested
- ➢ XML attribute values must be quoted

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
    <to>Tove</to>
    <From>Jani</from>
    <heading>Reminder</pheading>
    <body>Don't forget me this weekend!</body>
</note>
```

■ Valid Documents

○ A **well formed** XML document is not the same as a **valid** XML document

- ➢ DTD - The original Document Type Definition
- ➢ XML Schema - An XML-based alternative to DTD

■ DTD

○ define the structure of an XML document

```
<!DOCTYPE note SYSTEM "Note.dtd">
```

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

○ With a DTD

- ➢ You can use a DTD to verify your own data
- ➢ You can also verify that the data you receive from the outside world is valid

○ XML does not require a DTD/Schema

- ➢ when you are working with small XML files, creating DTDs may be a waste of time

■ XML Schema

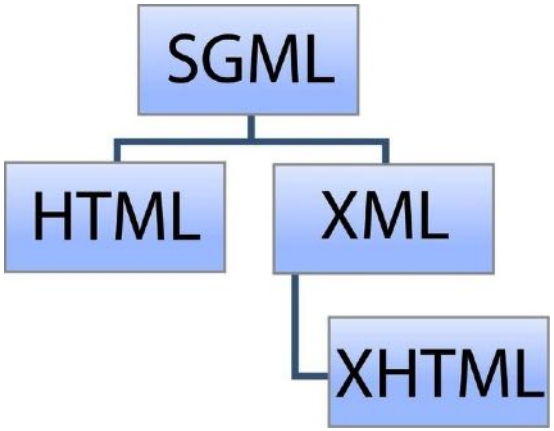
○ XML Schema Definition is an XML-based alternative to DTD

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

<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

○ XML Schemas are More Powerful than DTD

- XML Schemas are written in XML
- XML Schemas are extensible to additions
- XML Schemas support data types
- XML Schemas support namespaces

■ XML vs HTML



XML

```
<firstName>Maria</firstName>
<lastName>Roberts</lastName>
<dateBirth>12-11-1942</dateBirth>
```

HTML

```
<font size="3">Maria Roberts</font>
<b>12-11-1942</b>
```

○ Comparison

- XML was designed to carry data - with focus on what data is
- HTML was designed to display data - with focus on how data looks
- XML tags are not predefined like HTML tags are

COMPARISON

XML	HTML
•Extensible set of tags	•Fixed set of tags
•Content orientated	•Presentation oriented
•Standard Data infrastructure	•No data validation capabilities
•Allows multiple output forms	•Single presentation
•Content and format can be placed together.	•Content and format are separate; formatting is contained in a style sheet.

■ XML vs JSON

- JSON is lightweight thus simple to read and write.
- JSON supports array data structure.
- JSON files are more human readable.
- JSON has no display capabilities .
- Provides scalar data types and the ability to express structured data through arrays and objects.
- Native object support.

- XML is less simple than JSON.
- XML doesn't support array data structure.
- XML files are less human readable.
- XML provides the capability to display data because it is a markup language.
- Does not provide any notion of data types. One must rely on XML Schema for adding type information.
- Objects have to be expressed by conventions, often through a mixed use of attributes and elements.

Similarities between JSON and XML

Both are simple and open.

Both supports unicode. So internationalization is supported by JSON and XML both.

Both represents self describing data.

Both are interoperable or language-independent.

■ XML vs JSON

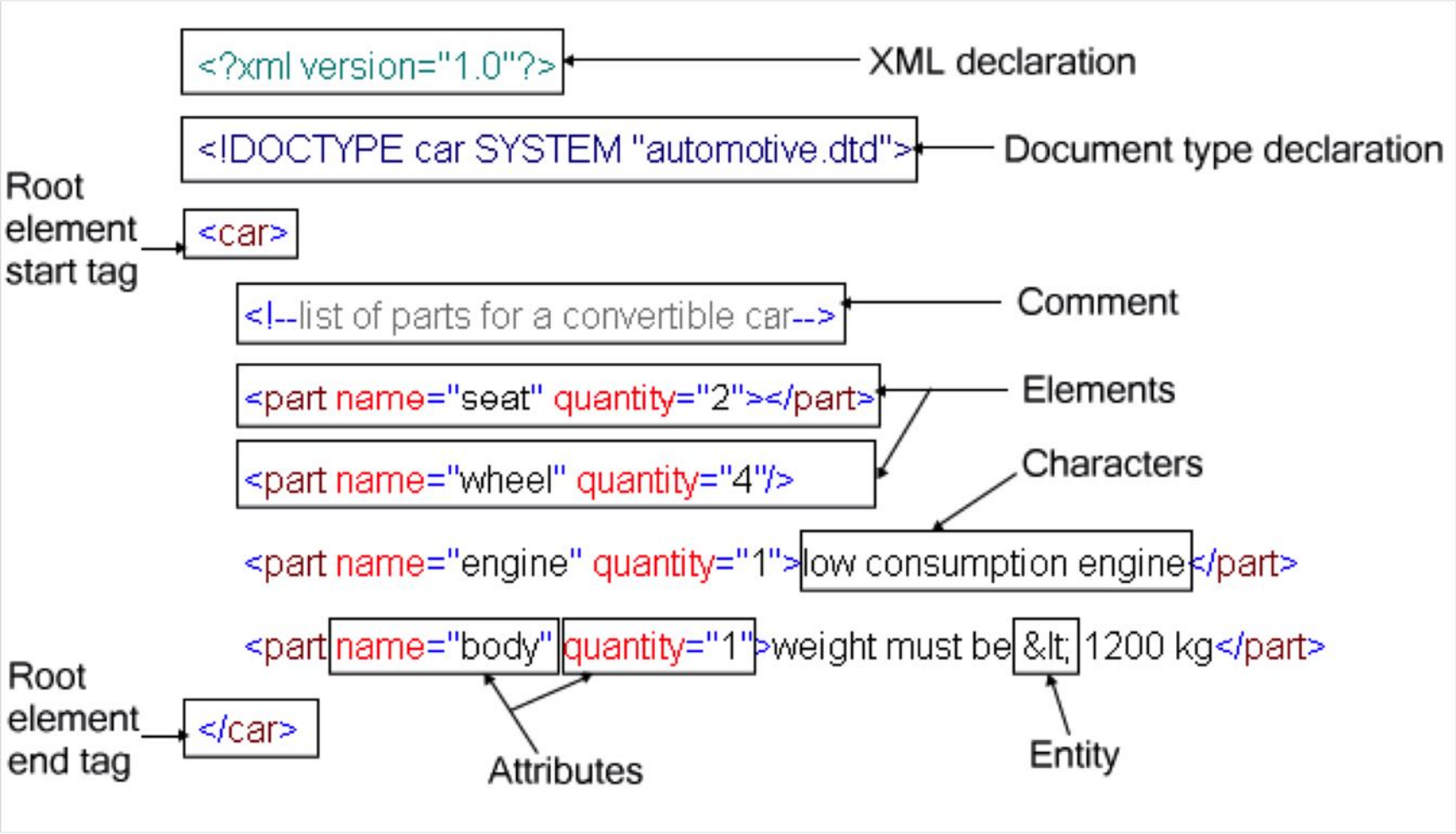
XML

```
<empinfo>
   <employees>
      <employee>
         <name>James Kirk</name>
         <age>40></age>
      </employee>
      <employee>
         <name>Jean-Luc Picard</name>
         <age>45</age>
      </employee>
      <employee>
         <name>Wesley Crusher</name>
         <age>27</age>
      </employee>
   </employees>
</empinfo>
```

JSON

```
{  "empinfo" :
      {
            "employees" :  [
            {
                  "name" : "James Kirk",
                  "age" : 40,
            },
            {
                  "name" : "Jean-Luc Picard",
                  "age" : 45,
            },
            {
                  "name" : "Wesley Crusher",
                  "age" : 27,
            }
                              ]
      }
}
```

■ Syntax


<?xml version="1.0"?>
XML declaration
<!DOCTYPE car SYSTEM "automotive.dtd">
Document type declaration
Root element start tag
<car>
<!--list of parts for a convertible car-->
Comment
<part name="seat" quantity="2"></part>
Elements
<part name="wheel" quantity="4"/>
Characters
<part name="engine" quantity="1">low consumption engine</part>
<part name="body" quantity="1">weight must be < 1200 kg</part>
Root element end tag
</car>
Attributes
Entity

○ XML Prolog

- The XML prolog is optional. If it exists, it must come first in the document
- UTF-8 is the default character encoding for XML documents

○ Closing Tag

- All elements **must** have a closing tag

○ Case Sensitive

- Opening and closing tags must be written with the same case

○ Properly Nested

- all elements **must** be properly nested within each other

```
<b><i>This text is bold and italic</b></i>
```

○ Quote

- the attribute values must always be quoted

○ Entity References

&lt;	<	less than
&gt;	>	greater than
&amp;	&	ampersand
&apos;	'	apostrophe
&quot;	"	quotation mark

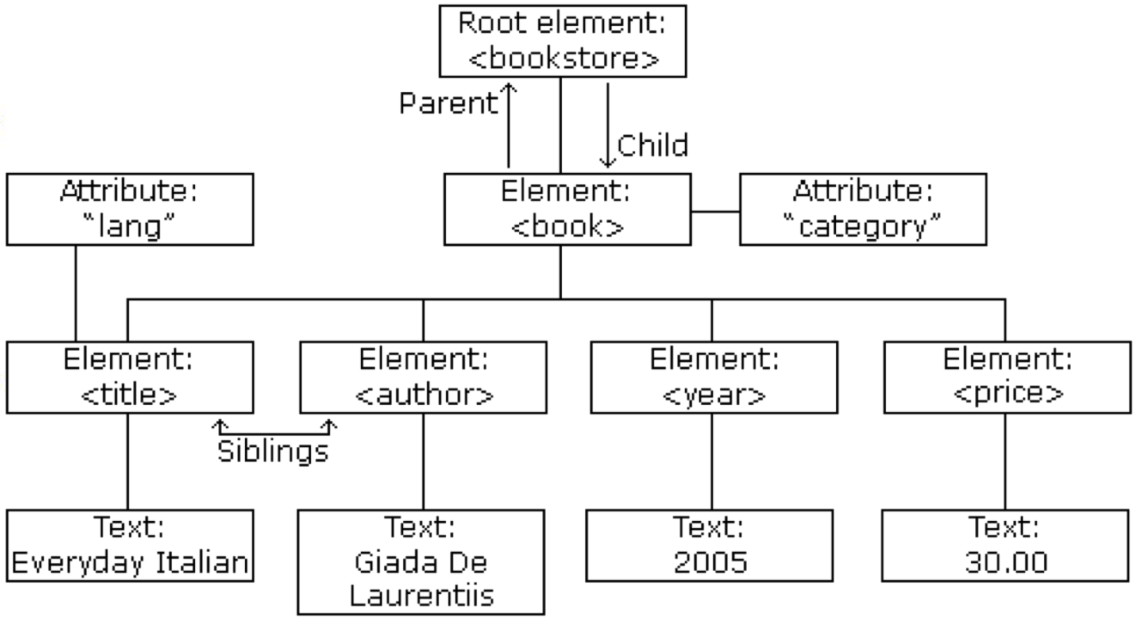
■ XML Tree

○ XML documents are formed as element trees.

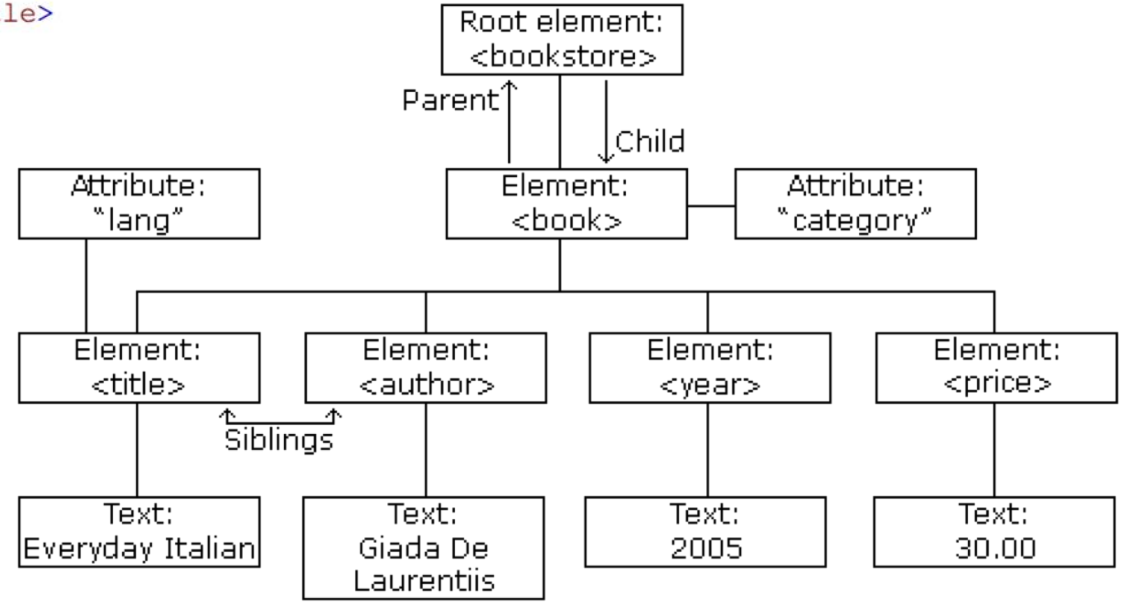
○ An XML tree starts at a **root** element and branches from the root to **child** elements

○ The terms **parent**, **child**, and **sibling** are used to describe the **relationships** between elements

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```



```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```



■ XML Elements

○ Everything from (including) the element's start tag to (including) the element's end tag

○ Naming rules

- Element names are case-sensitive
- Element names must start with a letter or underscore
- Element names cannot start with the letters xml (or XML, or Xml, etc)
- Element names can contain letters, digits, hyphens, underscores, and periods
- Element names cannot contain spaces

Style	Example	Description
Lower case	<firstname>	All letters lower case
Upper case	<FIRSTNAME>	All letters upper case
Underscore	<first_name>	Underscore separates words
Pascal case	<FirstName>	Uppercase first letter in each word
Camel case	<firstName>	Uppercase first letter in each word except the first

Avoid "-". If you name something "first-name", some software may think you want to subtract "name" from "first".

Avoid ".". If you name something "first.name", some software may think that "name" is a property of the object "first".

Avoid ":". Colons are reserved for namespaces.

Non-English letters like éòá are perfectly legal in XML, but watch out for problems if your software doesn't support them.

■ XML Attributes

○ Attributes are designed to contain data related to a specific element

○ Attribute values must always be quoted. Either single or double quotes can be used

```
<person gender="female">
```

```
<gangster name='George "Shotgun" Ziegler'>
```

■ XML Elements vs Attributes

○ There are no rules about when to use attributes or when to use elements in XML

```
<person gender="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

```
<person>
  <gender>female</gender>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

■ Namespace

○ XML Namespaces provide a method to avoid element name conflicts

```
<table>
  <tr>
    <td>Apples</td>
    <td>Bananas</td>
  </tr>
</table>
```

HTML table information

```
<table>
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

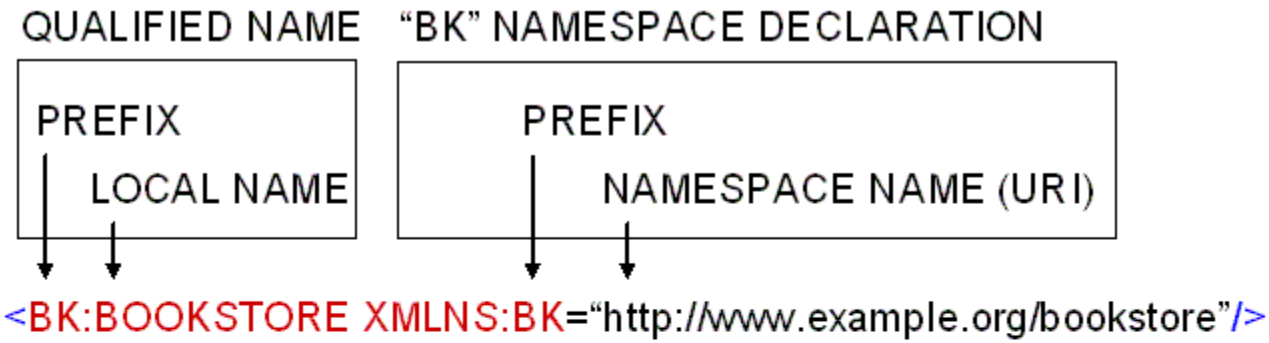
information about a table (a piece of furniture)

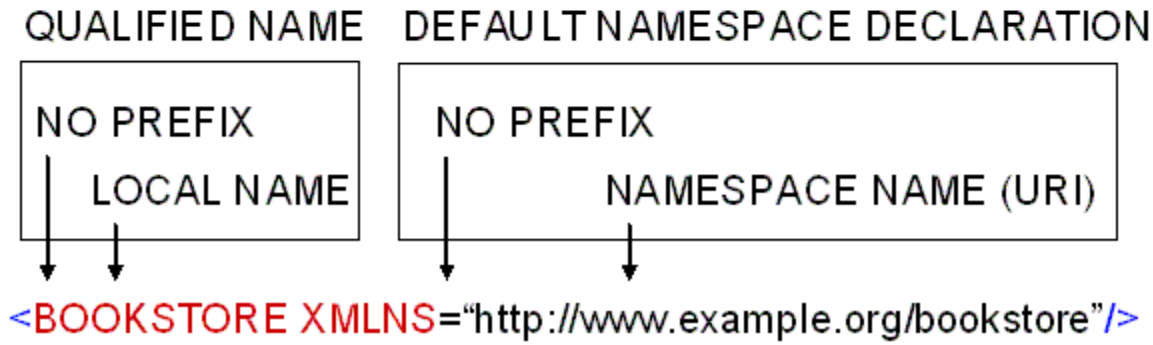
○ Solving the name conflict

```
<h:table>
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

○ xmlns

- a namespace for the prefix must be defined
- the namespace can be defined by an xmlns attribute in the start tag of an element



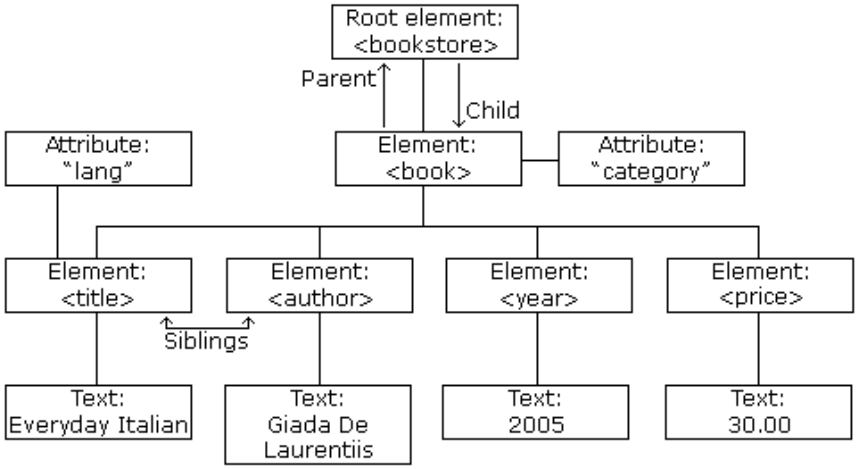


■ DOM

○ The DOM defines a standard for accessing and manipulating documents

> *"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."*

- A standard object model for XML
- A standard programming interface for XML
- Platform- and language-independent
- A W3C standard




```
getElementsByTagName("title")[0].childNodes[0].nodeValue
```

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

○ 예제

```
<body>
<p id="demo"></p>

<script>
  var parser, xmlDoc;
  var text = "\
      <bookstore>\
          <book>\
              <title>Everyday Italian</title>\
              <author>Giada De Laurentiis</author>\
              <year>2005</year>\
          </book>\
      </bookstore>";

  parser = new DOMParser();
  xmlDoc = parser.parseFromString(text,"text/xml");

  document.getElementById("demo").innerHTML =
  xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
</script>
</body>
```

■ DOM Parser

○ Methods

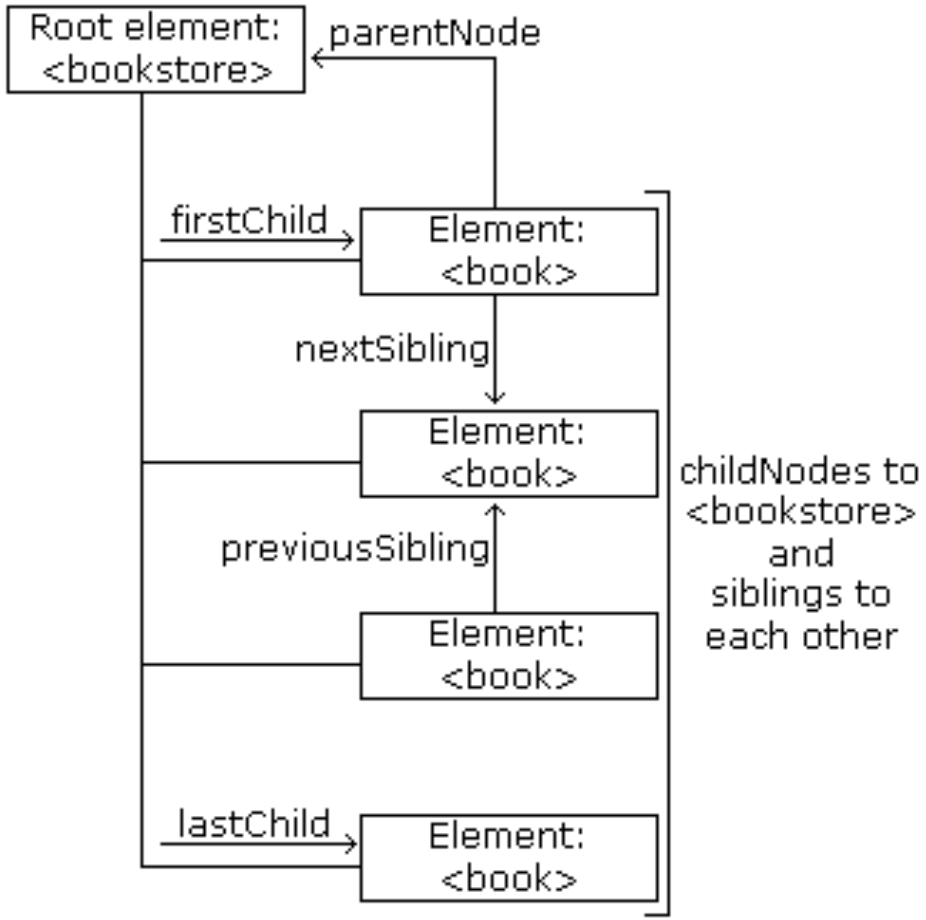
- x.getElementsByTagName(*name*) - get all elements with a specified tag name
- x.appendChild(*node*) - insert a child node to x
- x.removeChild(*node*) - remove a child node from x

○ Properties

- x.nodeName - the name of x
- x.nodeValue - the value of x
- x.parentNode - the parent node of x
- x.childNodes - the child nodes of x
- x.attributes - the attributes nodes of x

○ 예제

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">XQuery Kick Start</title>
    <author>James McGovern</author>
    <author>Per Bothner</author>
    <author>Kurt Cagle</author>
    <author>James Linn</author>
    <author>Vaidyanathan Nagarajan</author>
    <year>2003</year>
    <price>49.99</price>
  </book>
  <book category="web" cover="paperback">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```



○ Avoid empty text node

- Firefox, and some other browsers, will treat empty white-spaces or new lines as text nodes
- Internet Explorer will not

○ DOM Navigating

```
<script>
  var xhttp = new XMLHttpRequest();

  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      var xmlDoc = this.responseXML;

      bookList = xmlDoc.getElementsByTagName("book");

      console.log(bookList[0].parentNode.childNodes);

      console.log(bookList[0].parentNode);

      console.log(bookList[0].parentNode.firstChild.nextSibling);

      console.log(bookList[0].parentNode.childNodes[3]);

      console.log(bookList[0].parentNode.childNodes[3].previousSibling.previousSibling);

      console.log(bookList[0].parentNode.lastChild.previousSibling);
    }
  };

  xhttp.open("GET", "books.xml", true);
  xhttp.send();
</script>
```

○ DOM handling

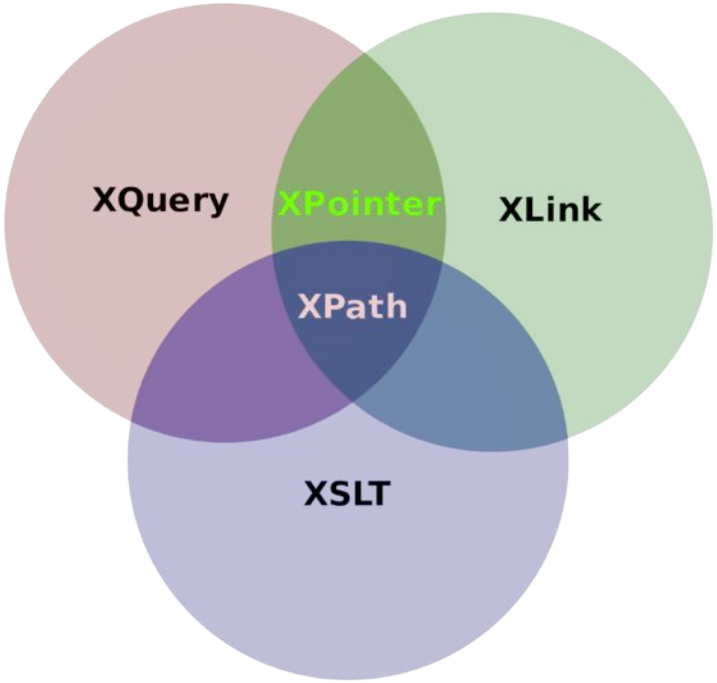
```
<script>
  var xhttp = new XMLHttpRequest();

  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      var xmlDoc = this.responseXML;

      bookList = xmlDoc.getElementsByTagName("book");

      console.log(bookList[0].childNodes[1]);
      bookList[0].childNodes[1].firstChild.nodeValue = "Everyday Korean";
      console.log(bookList[0].childNodes[1]);

      bookList[0].setAttribute("category", "history");
      console.log(bookList[0].getAttribute("category"));

      console.log(bookList);
      xmlDoc.documentElement.removeChild(bookList[0]);
      console.log(bookList);

      newNode = xmlDoc.createElement("edition");
      newText = xmlDoc.createTextNode("first");
      newNode.appendChild(newText);
      console.log(newNode);

      bookList[0].appendChild(newNode);
      console.log(bookList[0]);

      console.log(bookList[0].lastChild);
    }
  };

  xhttp.open("GET", "books.xml", true);
  xhttp.send();
</script>
```

■ XPath

○ XPath is a major element in the XSLT standard.

○ XPath can be used to navigate through elements and attributes in an XML document

- XPath is a syntax for defining parts of an XML document
- XPath uses path expressions to navigate in XML documents
- XPath contains a library of standard functions
- XPath is a major element in XSLT and in XQuery
- XPath is a W3C recommendation



○ Syntax

Expression	Description
nodename	Selects all nodes with the name "*nodename*"
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

Path Expression	Result
bookstore	Selects all nodes with the name "bookstore"
/bookstore	Selects the root element bookstore **Note:** If the path starts with a slash (/) it always represents an absolute path to an element!
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
//@lang	Selects all attributes that are named lang

○ 예제

XPath Expression	Result
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='en']	Selects all the title elements that have a "lang" attribute with a value of "en"
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

➢ Chrome, Firefox, Edge, Opera, and Safari

```
xmlDoc.evaluate(xpath, xmlDoc, null, XPathResult.ANY_TYPE,null);
```

○ 예제

```
<script>
  var xhttp = new XMLHttpRequest();

  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      var xmlDoc = this.responseXML;

      pathList = ["/bookstore/book[1]",
                  "/bookstore/book[last()]",
                  "/bookstore/book[last()-1]",
                  "/bookstore/book[position()<3]",
                  "//title[@lang]",
                  "//title[@lang='en']",
                  "/bookstore/book[price>35.00]",
                  "/bookstore/book[price>35.00]/title"];


      for (i=0; i<pathList.length; i++) {
        console.log((i+1), pathList[i]);

        var nodes = xmlDoc.evaluate(pathList[i], xmlDoc, null, XPathResult.ANY_TYPE, null);

        while (result = nodes.iterateNext()) {
          console.log(result);
        }
      }
    }
  };

  xhttp.open("GET", "books.xml", true);
  xhttp.send();
</script>
```

■ XML with Python

```
//input
[@name='token']
/@value
```

python™

○ XML library

- https://wiki.python.org/moin/PythonXml

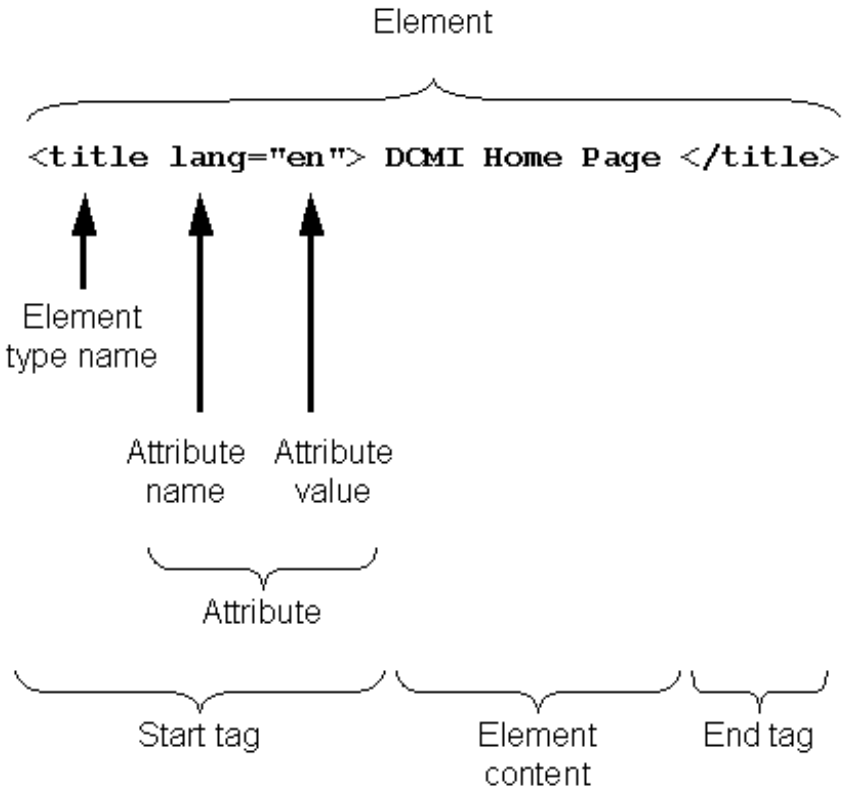
○ xml

```
import xml.etree.ElementTree as et
```

○ lxml

```
from lxml import etree
```

■ Building XML



○ Element

class `xml.etree.ElementTree.`**`Element`**(*tag, attrib={}, **extra*)
Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *tag* is the element name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments.

➢ append

append(*subelement*)
Adds the element *subelement* to the end of this elements internal list of subelements.

➢ insert

insert(*index*, *element*)
Inserts a subelement at the given position in this element.

➢ set

set(*key*, *value*)
Set the attribute *key* on the element to *value*.

➢ get

get(*key*, *default=None*)
Gets the element attribute named *key*.

○ SubElement

`xml.etree.ElementTree.` **SubElement**(*parent*, *tag*, *attrib={}*, ***extra*)
Subelement factory. This function creates an element instance, and appends it to an existing element.

○ 예제

```
bookStore = et.Element("bookstore")

book1 = et.Element("book", category="cooking")
bookStore.insert(0, book1)

title1 = et.Element("title")
title1.attrib["lang"] = "en"
title1.text = "Everyday Italian"
book1.append(title1)

et.SubElement(book1, "author").text = "Giada De Laurentiis"
et.SubElement(book1, "year").text = "2005"
et.SubElement(book1, "price").text = "30.00"

book2 = et.Element("book", {"category":"children"})
bookStore.append(book2)

title2 = et.Element("title")
title2.attrib["lang"] = title1.get("lang")
title2.text = "Harry Potter"
book2.append(title2)

et.SubElement(book2, "author").text = "Giada De Laurentiis"
et.SubElement(book2, "year").text = "2005"
et.SubElement(book2, "price").text = "30.00"

et.dump(bookStore)
```

○ dump

xml.etree.ElementTree.**dump**(*elem*)

Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

■ Parsing

○ parse

`xml.etree.ElementTree.`**`parse`**(*source, parser=None*)

Parses an XML section into an element tree.
source is a filename or file object containing XML data.
parser is an optional parser instance. If not given, the standard `XMLParser` parser is used.
Returns an `ElementTree` instance.

○ XML, fromstring

`xml.etree.ElementTree.`**`XML`**(*text, parser=None*)

Parses an XML section from a string constant.
This function can be used to embed "XML literals" in Python code.
text is a string containing XML data.
parser is an optional parser instance. If not given, the standard `XMLParser` parser is used.
Returns an `Element` instance.

`xml.etree.ElementTree.`**`fromstring`**(*text*)

Parses an XML section from a string constant. Same as `XML()`.
text is a string containing XML data. Returns an `Element` instance.

○ Element

➢ getchildren

➢ items

items()
Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

➢ keys

keys()
Returns the elements attribute names as a list. The names are returned in an arbitrary order.

➢ find

find(*match*)
Finds the first subelement matching *match*. *match* may be a tag name or path. Returns an element instance or `None`.

➢ findall

findall(*match*)
Finds all matching subelements, by tag name or path. Returns a list containing all matching elements in document order.

➢ findtext

findtext(*match*, *default=None*)
Finds text for the first subelement matching *match*. *match* may be a tag name or path.
Returns the text content of the first matching element, or *default* if no element was found.
Note that if the matching element has no text content an empty string is returned.

○ 예제

```
root = et.XML(et.tostring(bookStore))

# self.children
print(len(root))
for childNode in root:
    print(childNode.tag, childNode.attrib)

root.clear()

root = et.fromstring(et.tostring(bookStore))

# self.children, list(elem)
childNodes = root.getchildren()
print(len(childNodes))
for childNode in childNodes:
    print(childNode.tag, childNode.items())

for childNode in childNodes[0]:
    print(childNode.tag, childNode.keys())
    if childNode.keys() != []:
        print([childNode.get(k) for k in childNode.keys()])

book = root.find("book")
print(book.tag, book.get("category"))

bookList = root.findall("book")
for book in bookList:
    print(book.tag, book.get("category"))

title = root.find(".//title")
print(type(title), title.text)

titleList = root.findall(".//title")
print([title.text for title in titleList])

title = root.findtext(".//title")
print(type(title), title)

book = root.find(".//book[@category='children']")
print(book, book.tag)
```

■ XML File

○ write

write(*file, encoding="us-ascii", xml_declaration=None, default_namespace=None, method="xml"*)

Writes the element tree to a file, as XML. *file* is a file name, or a file object opened for writing.
encoding [1] is the output encoding (default is US-ASCII).
xml_declaration controls if an XML declaration should be added to the file.
Use `False` for never, `True` for always, `None` for only if not US-ASCII or UTF-8 (default is `None`).
default_namespace sets the default XML namespace (for "xmlns").
method is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). Returns an encoded string.

○ parse

parse(*source, parser=None*)

Loads an external XML section into this element tree
source is a file name or file object.
parser is an optional parser instance. If not given, the standard XMLParser parser is used.
Returns the section root element.

○ ElementTree

class `xml.etree.ElementTree.`**ElementTree**(*element=None, file=None*)

ElementTree wrapper class.
This class represents an entire element hierarchy,
and adds some extra support for serialization to and from standard XML.

○ 예제

➢ write

```
from xml.etree.ElementTree import ElementTree

tree = ElementTree(root)
tree.write("book_xml.xml", encoding="utf-8", xml_declaration="utf-8")
```

➢ parse

```
from xml.etree.ElementTree import parse

tree = parse("book_xml.xml")
root = tree.getroot()

for node in root.iter():
    print(node.tag, node.text)
```

➢ ElementTree

```
tree = ElementTree(file="book_xml.xml")
root = tree.getroot()

for node in root.iter():
    print(node.tag, node.text)
```

○ iter

iter(*tag=None*)

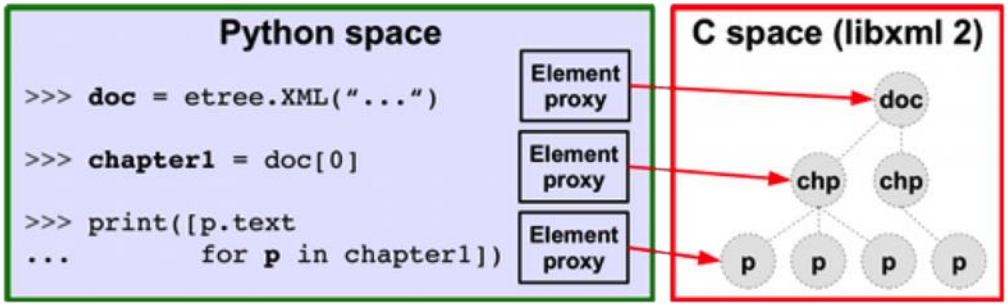
Creates a tree iterator with the current element as the root.
The iterator iterates over this element and all elements below it, in document (depth first) order.
If *tag* is not `None` or `'*'`, only elements whose tag equals *tag* are returned from the iterator.
If the tree structure is modified during iteration, the result is undefined.

■ lxml

lxml - XML and HTML with Python

lxml is the most feature-rich and easy-to-use library for processing XML and HTML in the Python language.

○ Being based on libxml2, lxml.etree holds the entire XML tree in a C structure



■ Element

*Element(_tag, attrib=None, nsmap=None, **_extra)*

Element factory. This function returns an object implementing the Element interface.

An `Element` is the main container object for the ElementTree API. Most of the XML tree functionality is accessed through this class. Elements are easily created through the `Element` factory:

○ SubElement

SubElement(_parent, _tag, attrib=None, nsmap=None, **_extra)
Subelement factory. This function creates an element instance, and appends it to an existing element.

○ tostring

tostring(element_or_tree, encoding=None, method="xml", xml_declaration=None, pretty_print=False, with_tail=True, standalone=None, doctype=None, exclusive=False, with_comments=True, inclusive_ns_prefixes=None)

Serialize an element to an encoded string representation of its XML tree.

Defaults to ASCII encoding without XML declaration. This behaviour can be configured with the keyword arguments 'encoding' (string) and 'xml_declaration' (bool). Note that changing the encoding to a non UTF-8 compatible encoding will enable a declaration by default.

You can also serialise to a Unicode string without declaration by passing the `unicode` function as encoding (or `str` in Py3), or the name 'unicode'. This changes the return value from a byte string to an unencoded unicode string.

The keyword argument 'pretty_print' (bool) enables formatted XML.

The keyword argument 'method' selects the output method: 'xml', 'html', plain 'text' (text content without tags) or 'c14n'. Default is 'xml'.

The `exclusive` and `with_comments` arguments are only used with C14N output, where they request exclusive and uncommented C14N serialisation respectively.

Passing a boolean value to the `standalone` option will output an XML declaration with the corresponding `standalone` flag.

The `doctype` option allows passing in a plain string that will be serialised before the XML tree. Note that passing in non well-formed content here will make the XML output non well-formed. Also, an existing doctype in the document tree will not be removed when serialising an ElementTree instance.

You can prevent the tail text of the element from being serialised by passing the boolean `with_tail` option. This has no impact on the tail text of children, which will always be serialised.

○ tounicode

tounicode(element_or_tree, method="xml", pretty_print=False, with_tail=True, doctype=None)
Serialize an element to the Python unicode representation of its XML tree.

○ dump

dump(elem, pretty_print=True, with_tail=True)
Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

○ 예제

```
bookStore = etree.Element("bookstore")

book1 = etree.SubElement(bookStore, "book")
book2 = etree.SubElement(bookStore, "book", attrib={"category":"children"})

book1.attrib["category"] = "cooking"

title1 = etree.Element("title", lang="en")
title1.text = "Everyday Italian"
book1.append(title1)

etree.SubElement(book1, "author").text = "Giada De Laurentiis"
etree.SubElement(book1, "year").text = "2005"
etree.SubElement(book1, "price").text = "30.00"

title2 = etree.Element("title")
title2.set("lang", title1.get("lang"))
title2.text = "Harry Potter"
book2.append(title2)

etree.SubElement(book2, "author").text = "Giada De Laurentiis"
etree.SubElement(book2, "year").text = "2005"
book2.insert(3, etree.Element("price"))

print(len(book2))
book2[-1].text = "30.00"

xmlBytes = etree.tostring(bookStore, encoding="utf-8", pretty_print=True, xml_declaration=True)
xmlStr = etree.tounicode(bookStore, pretty_print=True)
print(type(xmlBytes), type(xmlStr))
etree.dump(bookStore)
```

■ Parsing

○ parse

parse(source, parser=None, base_url=None)

Return an ElementTree object loaded with source elements. If no parser is provided as second argument, the default parser is used.

The `source` can be any of the following:

- a file name/path
- a file object
- a file-like object
- a URL using the HTTP or FTP protocol

○ XML, fromstring

XML(text, parser=None, base_url=None)

Parses an XML document or fragment from a string constant. Returns the root node (or the result returned by a parser target). This function can be used to embed "XML literals" in Python code, like in

fromstring(text, parser=None, base_url=None)

Parses an XML document or fragment from a string. Returns the root node (or the result returned by a parser target).

To override the default parser with a different parser you can pass it to the `parser` keyword argument.

The `base_url` keyword argument allows to set the original base URL of the document to support relative Paths when looking up external entities (DTD, XInclude, ...).

○ ElementTree

```
ElementTree(element=None, file=None, parser=None)
ElementTree wrapper class.
```

➢ 예제

```
xml = etree.XML(etree.tostring(bookStore))
xmlTree = etree.ElementTree(xml)
xmlRoot = xmlTree.getroot()

print(xmlTree.docinfo.xml_version)
print(xmlTree.docinfo.encoding)
print(xmlTree.docinfo.doctype)
print(xmlTree.docinfo.root_name)

print(len(xmlRoot))
for childNode in xmlRoot:
    print(childNode.tag, childNode.attrib)
```

○ 예제

```
xml = etree.fromstring(etree.tostring(bookStore))
xmlTree = etree.ElementTree(xml)
xmlRoot = xmlTree.getroot()

childNodes = xmlRoot.getchildren()

print(len(childNodes))
for childNode in childNodes:
    print(childNode.tag, childNode.items())

for childNode in childNodes[0]:
    print(childNode.tag, childNode.keys())
    if childNode.keys() != []:
        print([childNode.get(k) for k in childNode.keys()])

book = xmlRoot.find("book")
print(book.tag, book.get("category"))

bookList = xmlRoot.findall("book")
for book in bookList:
    print(book.tag, book.get("category"))

title = xmlRoot.find(".//title")
print(type(title), title.text)

titleList = xmlRoot.findall(".//title")
print([title.text for title in titleList])

title = xmlRoot.findtext(".//title")
print(type(title), title)

book = xmlRoot.find(".//book[@category='children']")
print(book, book.tag)

for childNode in xmlRoot.iter():
    print(childNode.tag, childNode.text)

for childNode in xmlRoot.iter("book"):
    print(childNode.tag, childNode.text)
```

○ 예제

➢ write

```
xmlTree.write("book_tree.xml")
etree.ElementTree(xmlRoot).write("book_root.xml")
```

➢ parse

```
xmlTree = etree.parse("book_tree.xml")
xmlRoot = xmlTree.getroot()

etree.dump(xmlRoot)

xmlTree = etree.parse("book_root.xml")
xmlRoot = xmlTree.getroot()

etree.dump(xmlRoot)
```

○ 전국 대기오염도 현황 Open API

- 서비스
  - https://www.data.go.kr/subMain.jsp#/L3B1YnIvcG90L215cC9Jcm9zTXlQYWdlL29wZW5EZXZHdWlkZVBhZ2UkQF4wMTIkQF5wdWJsaWNEYXRhRGV0YWlsUGs9dWRkaTo3MDkxMTBlNy1kN2IxLTQ0MjEtOTBiYS04NGE2OWY5ODBjYWJfMjAxNjA4MDgxMTE0JEBebWFpbkZsYWc9dHJ1ZQ==
- URL
  - http://openapi.airkorea.or.kr/openapi/services/rest/ArpltnInforInqireSvc/getMsrstnAcctoRltmMesureDnsty
- Parameters
  - stationName
  - dateTerm = *daily* | *month* | *3month*
  - pageNo
  - numOfRows
  - ver = *1.1* | *1.2* | *1.3* | *1.4*
  - _returnType = *json*

○ 전국 대기오염도 현황 Open API 활용

```
#resStr = resStr.decode("utf-8")
xmlObj = etree.fromstring(resStr)
xmlRoot = etree.ElementTree(xmlObj).getroot()

etree.dump(xmlRoot)

for node in xmlRoot.iter():
    print(node.tag, node.text)

itemList = xmlRoot.findall(".//item")

print(len(itemList))
for item in itemList:
    for i in range(len(item)):
        print(item[i].tag, item[i].text)

pm25List = xmlRoot.findall(".//item/pm25Value")
for item in pm25List:
    print(item.tag, item.text)
#         print(item.get(i).tag)
#         etree.dump(item.get(i))
```