# Importing Data into Python from JSON & XML Files

**Xavier Morera**
BIG DATA INC.

@xmorera www.xaviermorera.com

# In Previous Modules...


Creative Commons
Attribution-ShareAlike 3.0 Unported
CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM


```
1.0000000000000000e+00 1.0000000000000000e+00
2.0000000000000000e+00 2.0000000000000000e+00
3.0000000000000000e+00 4.0000000000000000e+00
```


```
-1,1,"on the server farm","Community"
1,101,"New York, NY","Adam Lear"
2,101,"Corvallis, OR","Geoff Dalgas
```

pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$

```
-1    1     "on the server farm"    "Community"
1     101   "New York, NY"    "Adam Lear"
2     101   "Corvallis       OR"       "Geoff Dalgas
```

# What Do You See Here?

-1,1,"on the server farm","Community"
1,101,"SJ, Costa Rica","Xavier"
2,101,"Barlovento, CR","Irene"

```
<?xml version="1.0" encoding="UTF-8" ?>
	<users>
		<id>-1</id>
		<reputation>1</reputation>
		<info>
			<location>on the server farm</location>
			<displayname>Community</displayname>
		</info>
	</users>
```

```
{
  "users": {
    "id": -1,
    "reputation": "1",
    "info": {
      "location": "on the server farm",
      "displayname": "Community"
    }
  }
}
```

{

}

# JavaScript Object Notation

**Lightweight data-interchange format**

- Easy for humans to read and write
- Machines to parse and generate

# Introducing JSON

العربية Български 中文 Český Dansk Nederlands English Esperanto Français Deutsch Ελληνικά עברית Magyar Indonesia
Italiano 日本 한국어 فارسی Polski Português Română Русский Српско-хрватски Slovenščina Español Svenska Türkçe Tiếng Việt

ECMA-404 The JSON Data Interchange Standard.

**JSON** (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

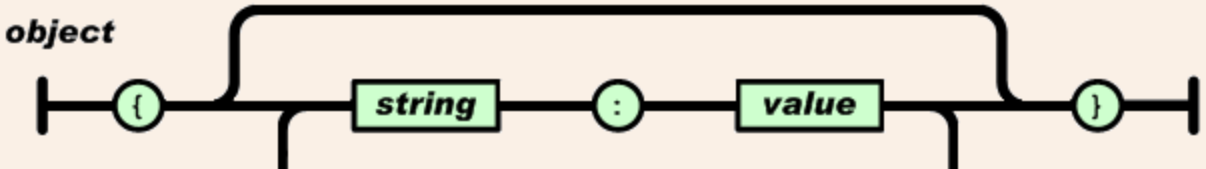
JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

In JSON, they take on these forms:

An *object* is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma).




```
json
    element

value
    object
    array
    string
    number
    "true"
    "false"
    "null"

object
    '{' ws '}'
    '{' members '}'

members
    member
    member ',' members

member
    ws string ws ':' element

array
    '[' ws ']'
    '[' elements ']'
```

```
{
  "users": {
    "id": -1,
    "reputation": "1",
    "info": {
      "location": "on the server farm",
      "displayname": "Community"
    }
  }
}
```

# JavaScript Object Notation

**Lightweight data-interchange format**

- Easy for humans to read and write
- Machines to parse and generate

**Python supports JSON natively, with the json library**

# `json` — JSON encoder and decoder

**Source code:** Lib/json/__init__.py

---

JSON (JavaScript Object Notation), specified by **RFC 7159** (which obsoletes **RFC 4627**) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript [1] ).

`json` exposes an API familiar to users of the standard library `marshal` and `pickle` modules.

Encoding basic Python object hierarchies:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

Compact encoding:

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

import json

# Importing JSON Data

**Start by importing the json library, and inspect the functions with help**

load

loads

```
# head posts-100.json
with open('posts-100.json', 'r') as f:
    posts_json = json.load(f)
type(posts_json)
len(posts_json)
len(posts_json[0])
print(posts_json[0]['Id'], posts_json[0]['Title'])
```

# Importing JSON Data with load

**Inspect your JSON**

**Use load to deserialize your data**

**Inspect your object**


```
{
  "Id": 5,
  "PostTypeId": "1",
  "CreationDate": "2014-05-13T23:58:30.457",

  "Score": 9,
  "ViewCount": 448,
  "LastActivityDate": "2014-05-14T00:36:31.077",
  "Title": "How can I do simple machine learning?",
  "Tags": "<machine-learning>",
  "AnswerCount": 1,
  "CommentCount": 1,
  "FavoriteCount": 1,
  "ClosedDate": "2014-05-14T14:40:25.950"
}
```


>>> posts_json[0]['"Title']
How can I do simple machine learning

```
json_original = """
{
  "Id": 5,
  "PostTypeId": "1",
  "CreationDate": "2014-05-13T23:58:30.457",
  "Score": 9,
  "ViewCount": 448,
  "LastActivityDate": "2014-05-14T00:36:31.077",
  "Title": "How can I do simple machine learning?",
  "Tags": "<machine-learning>",
  "AnswerCount": 1,
  "CommentCount": 1,
  "FavoriteCount": 1,
  "ClosedDate": "2014-05-14T14:40:25.950"
}

"""
```

```
json_loaded = json.loads(json_original)

json_loaded

type(json_loaded)

print(json.dumps(json_loaded))

print(json.dumps(json_loaded, indent=2))
```

# Importing JSON Data with loads

**Import JSON data in memory with loads**

**Work with your object**

- Serialize it again

# Translations on Loading

| JSON | Python |
| --- | --- |
| object | dict |
| array | list |
| string | str |
| number (int) | int |
| number (real) | float |
| true | True |
| false | False |
| null | None |

```
this_tuple = ('one', 'two')
this_list = ['one', 'two']
type(this_tuple)
type(this_list)
json_tuple = json.dumps(this_tuple)
json_list = json.dumps(this_list)
json_tuple
json_list
json_tuple == json_list
type(json.loads(json_tuple))
type(json.loads(json_list))
```

# A Few Things to Watch out For

**Potential changes when serializing-deserializing**

- Can affect the data you are loading

**Test with a tuple and a list**

```
weather_query = "https://query.yahooapis.com/v1/public/yql?q=select item.condition from weather.forecast
where woeid = 2487889&format=json&env=store://datatables.org/alltableswithkeys"

import requests

weather_call = requests.get(weather_query)

weather = json.loads(weather_call.text)

weather['query']['results']['channel']['item']['condition']['temp']
```

# Importing Data from a JSON API

**JSON is very popular for APIs**

**Import JSON data from one of them**

# json — JSON encoder and decoder

**Source code:** Lib/json/__init__.py

---

JSON (JavaScript Object Notation), specified by **RFC 7159** (which obsoletes **RFC 4627**) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript [1] ).

`json` exposes an API familiar to users of the standard library `marshal` and `pickle` modules.

Encoding basic Python object hierarchies:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

Compact encoding:

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
```

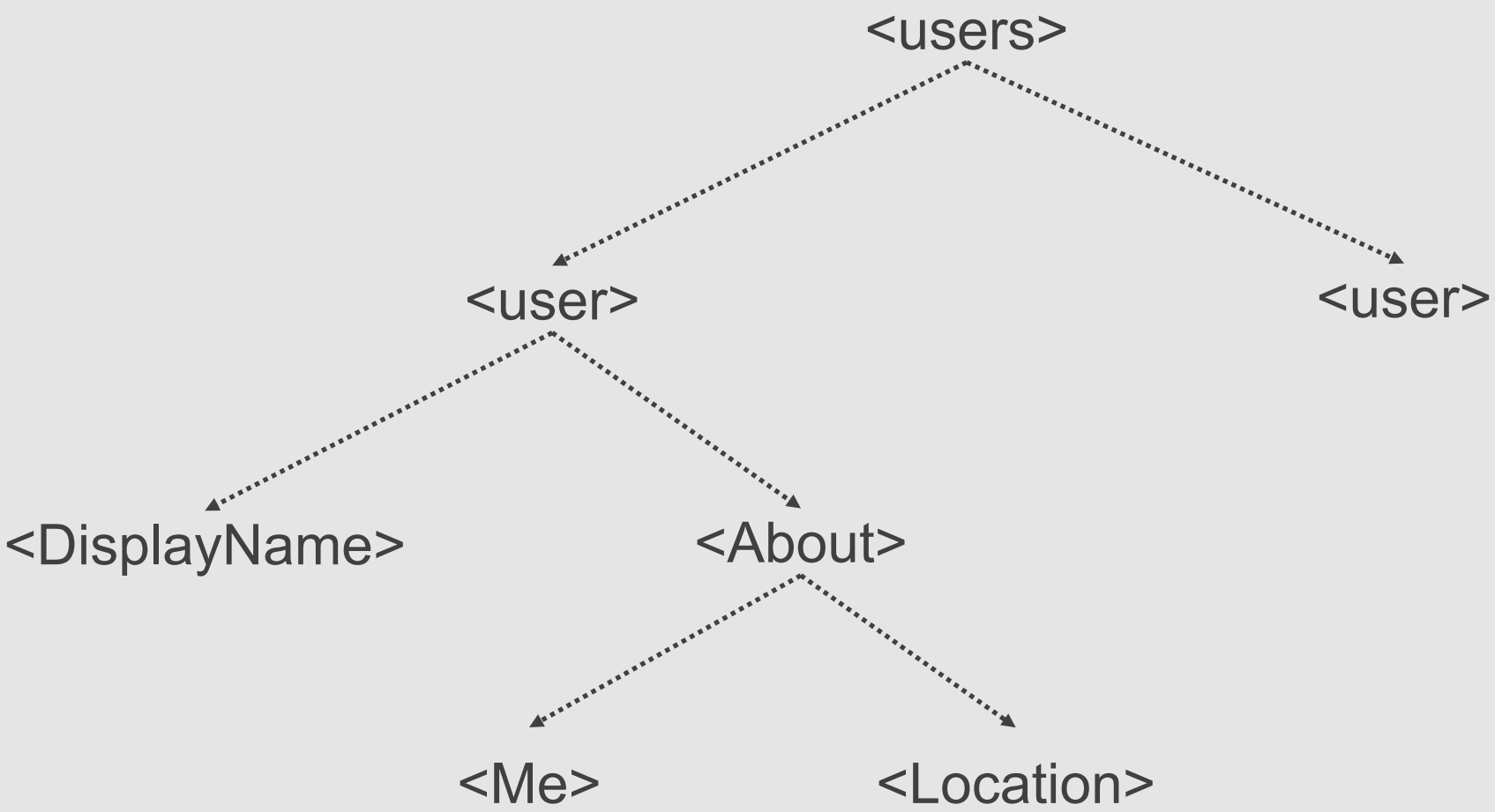
<?xml version='1.0'?>

# eXtensible Markup Language

**Data interchange format that's widely used and flexible**

- Store and transport data
- Human-readable, easy for machines to parse
- Data and metadata, in a hierarchical way

```
<?xml version='1.0'?>
<users>
	<user AccountId="16" CreationDate="2014-05-13T22:58:54.810"
		DownVotes="0" Id="1" LastAccessDate="2018-08-19T18:07:35.607"
		ProfileImageUrl="https://xaviermorera/xavier.jpg"
		Reputation="101" UpVotes="100" Views="256"
		WebsiteUrl="www.bigdatainc.org">
		<DisplayName>Xavier Morera</DisplayName>
		<About>
			<Me>Helping developers understand search and Big Data</Me>
			<Location>Costa Rica</Location>
		</About>
	</user>
</users>
```


<users>
<user>
<user>
<DisplayName>
<About>
<Me>
<Location>

```
import xml.etree.ElementTree as ET
```

# Importing XML with the ElementTree Module

## Start with the import

# xml.etree.ElementTree — The ElementTree XML API

**Source code:** Lib/xml/etree/ElementTree.py

---

The xml.etree.ElementTree module implements a simple and efficient API for parsing and creating XML data.

*Changed in version 3.3:* This module will use a fast implementation whenever available. The xml.etree.cElementTree module is deprecated.

**Warning:** The xml.etree.ElementTree module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

## Tutorial

This is a short tutorial for using xml.etree.ElementTree (ET in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

### XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two classes for this purpose - ElementTree represents the whole XML document as a tree, and Element represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the ElementTree level. Interactions with a single XML element and its sub-elements are done on the Element level.

### Parsing XML

```
# head –n2 users-100.xml

tree = ET.parse('users-100.xml')

tree
```

# Importing XML with the ElementTree Module

**Easy to load and process XML files**

**Create a tree structure using parse**

**Load into an ElementTree object**

```
users_root = tree.getroot()
users_root.tag
users_root.getchildren()
len(list(users_root.getchildren()))
users_root[0]
users_root[0].tag
users_root[0].attrib
```

# Importing XML with the ElementTree Module

**Need to get the root with getroot, and inspect**

**Now use getchildren**

- Review one Element

# xml.etree.ElementTree — The ElementTree XML API

**Source code:** Lib/xml/etree/ElementTree.py

---

The xml.etree.ElementTree module implements a simple and efficient API for parsing and creating XML data.

*Changed in version 3.3:* This module will use a fast implementation whenever available. The xml.etree.cElementTree module is deprecated.

**Warning:** The xml.etree.ElementTree module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

## Tutorial

This is a short tutorial for using xml.etree.ElementTree (ET in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

### XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two classes for this purpose - ElementTree represents the whole XML document as a tree, and Element represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the ElementTree level. Interactions with a single XML element and its sub-elements are done on the Element level.

### Parsing XML