

# JSON – JavaScript Object Notation

Some of the slides taken from  
Douglas Crockford / Paypal Inc.

This content is protected and may not be shared, uploaded, or distributed.

# What is JSON

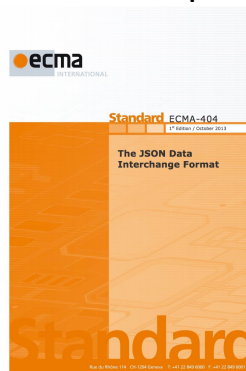
- **JSON**, short for **JavaScript Object Notation**, is a lightweight data interchange format.
  - It is a text-based, human-readable format for representing simple data structures and associative arrays (called objects).
- The JSON format is specified in RFC 4627 (July **2006**) authored by Douglas Crockford.
  - <https://tools.ietf.org/rfc/rfc4627.txt>
  - The official MIME type for JSON is application/json. The JSON file extension is .json.
- The JSON format is often used for transmitting structured data over a network connection in a process called serialization.
  - Its main application is in Ajax web application programming, where it serves as an alternative to the use of the XML format.
- Code for parsing and generating JSON data is readily available for a large variety of programming languages. The [www.json.org](http://www.json.org) website provides a comprehensive listing of existing JSON bindings, organized by language.

# Brief History

- JSON was based on a subset of the JavaScript programming language (specifically, Standard ECMA-262 - now in its 13th Edition)
  - however, it is a language-independent data format.
  - For the complete specification see

<https://www.ecma-international.org/publications/standards/Ecma-262.htm>

The JavaScript ECMA standard is based upon Netscape's JavaScript and Microsoft's Jscript



- Douglas Crockford was the original developer of JSON while he was at State Software, Inc. He is now Senior JavaScript Architect at Paypal
- <http://www.json.org/>, is a website devoted to JSON discussions and includes many JSON parsers

# How to use the JSON format

- A JSON file allows one to load data from the server or to send data to it.
- Working with JSON involves three steps: (i) the browser processing, (ii) the server processing, and (iii) the data exchange between them.

## 1. Client side (browser)

- The content of a JSON file (or stream), or the definition of JSON data is assigned to a variable, and this variable becomes an object of the program.

## 2. Server side

- a JSON file (or stream) on the server can be operated upon by various programming languages, including PHP and Java thanks to parsers that process the file and may even convert it into classes and attributes of the language.

## 3. Data exchange

- Loading a JSON file from the server may be accomplished in JavaScript in several ways:
  - directly including the file into the HTML page, as a JavaScript **.json external file**.
  - loading by a JavaScript command
  - using **XMLHttpRequest()**
- To convert JSON into an object, it can be passed to the JavaScript **eval()** function.
- Sending the file to the server may be accomplished by **XMLHttpRequest()**. The file is sent as a text file and processed by the parser of the programming language that uses it.

# JSON and XMLHttpRequest Example

- **XMLHttpRequest** will be covered in more detail in the AJAX lecture

- **The XMLHttpRequest code:**

```
var req = new XMLHttpRequest();  
req.open("GET", "file.json", true); // "asynchronous" operation  
req.onreadystatechange = myCode; // the callback  
req.send(null);
```

- **The JavaScript callback: eval() parses JSON, creates an object and assigns it to variable doc**

```
function myCode() {  
    if (req.readyState == 4) {  
        if (req.Status == 200) {  
            var doc = eval('(' + req.responseText + ')');  
        }  
    }  
}
```

- **Using the data:**

```
var menuName = doc.getElementById('menu'); // finding a field menu  
doc.menu.value = "my name is"; // assigning a value to the field
```

- **How to access data:**

```
doc.commands[0].title // read value of the "title" field in the array  
doc.commands[0].action // read value of the "action" field in the array
```

# JavaScript eval()

- The JavaScript **eval ()** is a function property of the Global Object, and evaluates a string and executes it as if it was JavaScript code, e.g.

```
<script type="text/javascript">
eval ("x=10;y=20;document.write(x*y)");
document.write("<br />");
document.write(eval("2+2"));
document.write("<br />");
var x=10;
document.write(eval(x+17));
document.write("<br />"); </script>
```

- produces the output

```
200
4
27
```

- Because JSON-formatted text is also syntactically legal JavaScript code, an easy way for a JavaScript program to parse JSON-formatted data is to use the built-in JavaScript eval() function
- the JavaScript interpreter itself is used to *execute* the JSON data to produce native JavaScript objects.
- The eval() technique is subject to security vulnerabilities if the data and the entire JavaScript environment is not within the control of a single trusted source; See:
- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/eval](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval)

# JSON Basic Data Types

- ***String*** (double-quoted unicode with backslash escaping)
- ***Numbers*** (integer, real, or floating point)
- ***Booleans*** (true and false)
- ***Object*** (collection of key:value pairs, comma-separated and enclosed in curly brackets)
- ***Array*** (an ordered sequence of values, comma-separated and enclosed in square brackets)
- ***Null*** (a value that isn't anything)

# String

- Sequence of 0 or more Unicode characters
- No separate character type
  - A character is represented as a string with a length of 1
- Wrapped in "double quotes"
- Backslash escapement



# Object

- Objects are unordered containers of key/value pairs
- Objects are wrapped in { }
- , separates key/value pairs
- : separates keys and values
- Keys are strings (unlike in JavaScript)
- Values are JSON values
- Can be used to represent struct, record, hashtable, object

# Example Object

```
{"name": "Jack B. Nimble", "at large":  
true, "grade": "A", "level": 3,  
"format": {"type": "rect", "width": 1920,  
"height": 1080, "interlace": false,  
"framerate": 24}}
```

# Example Object Formatted

```
{  
  "name":      "Jack B. Nimble",  
  "at large": true,  
  "grade":     "A",  
  "level":    3,  
  "format": {  
    "type":      "rect",  
    "width":     1920,  
    "height":    1080,  
    "interlace": false,  
    "framerate": 24  
  }  
}
```

# Array

- Arrays are ordered sequences of values
- Arrays are wrapped in [ ] (square brackets)
- , separates values
- JSON does not talk about indexing.
  - An implementation can start array indexing at 0 or 1.

# Two Examples of JSON Arrays

- **One dimensional**

```
["Sunday", "Monday", "Tuesday", "Wednesday",  
 "Thursday", "Friday", "Saturday"]
```

- **Two dimensional**

```
[  
  [0, -1, 0],  
  [1, 0, 0],  
  [0, 0, 1]  
]
```

# Arrays vs Objects

- Use objects when the key names are arbitrary strings
- Use arrays when the key names are sequential integers

# JSON is Not XML

## JSON

- Objects
- Arrays
- Strings
- Numbers
- Booleans
- null

## XML

- element
- attribute
- Attribute string
- content
- <![CDATA[ ]]>
- Entities
- Declarations
- Schema
- Stylesheets
- Comments
- Version
- namespace

# JSON vs. XML Example

## JSON

```
{
  "menu": "File",
  "commands": [
    {
      "title": "New",
      "action": "CreateDoc"
    },
    {
      "title": "Open",
      "action": "OpenDoc"
    },
    {
      "title": "Close",
      "action": "CloseDoc"
    }
  ]
}
```

## XML Equivalent

```
<?xml version="1.0" ?>
<root>
  <menu>File</menu>
  <commands>
    <item>
      <title>New</value>
      <action>CreateDoc</action>
    </item>
    <item>
      <title>Open</value>
      <action>OpenDoc</action>
    </item>
    <item>
      <title>Close</value>
      <action>CloseDoc</action>
    </item>
  </commands>
</root>
```

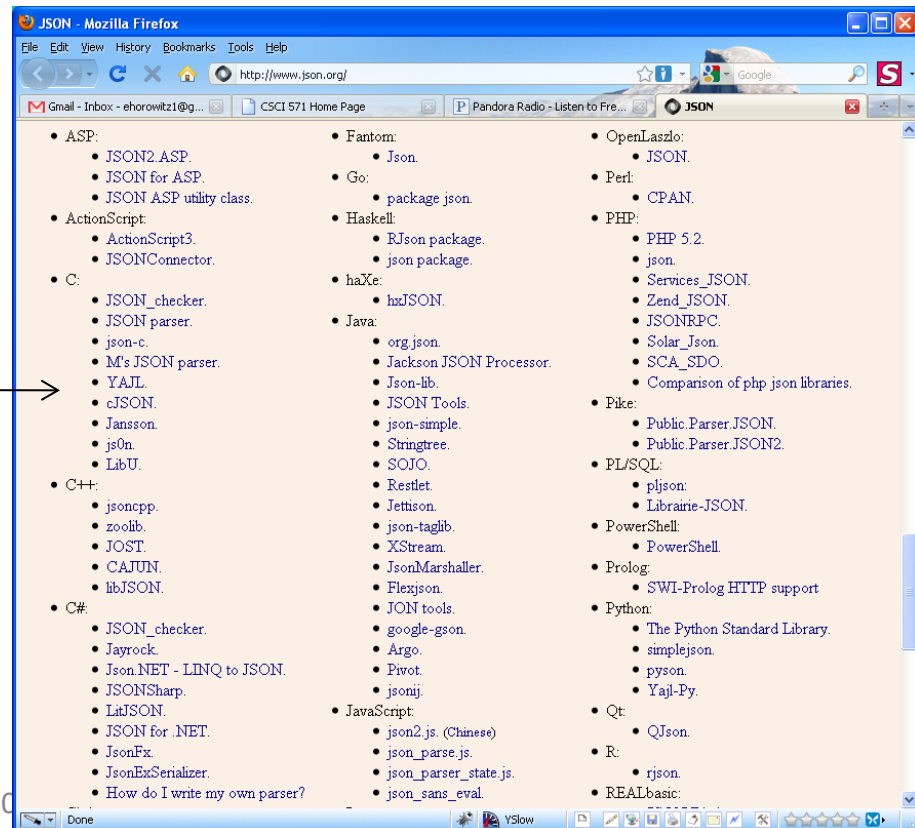
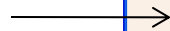


# Rules for JSON Parsers

- A JSON decoder must accept all well-formed JSON text
- A JSON decoder may also accept non-JSON text
- A JSON encoder must only produce well-formed JSON text
- A list of decoders for JSON can be found at

<http://www.json.org/>

JSON parsers for programming languages include C, C++, C#, Java, JavaScript, Perl, PHP



# Same Origin Policy

- Same origin policy is a security feature that browsers apply to client-side scripts
- It prevents a document or script loaded from one “origin” from getting or setting properties of a document from a different “origin”
  - Rationale: the browser should not trust content loaded from arbitrary websites
- Given the URL: <http://www.example.com/dir/page.html>

URL	Outcome	Reason
<a href="http://www.example.com/dir2/other.json">http://www.example.com/dir2/other.json</a>	Success	Same protocol and host
<a href="http://www.example.com/dir/inner/other.json">http://www.example.com/dir/inner/other.json</a>	Success	Same protocol and host
<a href="http://www.example.com:81/dir2/other.json">http://www.example.com:81/dir2/other.json</a>	Failure	Same protocol and host but different port
<a href="https://www.example.com/dir2/other.json">https://www.example.com/dir2/other.json</a>	Failure	Different protocol
<a href="http://en.example.com/dir2/other.json">http://en.example.com/dir2/other.json</a>	Failure	Different host
<a href="http://example.com/dir2/other.json">http://example.com/dir2/other.json</a>	Failure	Different host

# JSON: The Cross-Domain Hack

- JSON and the **<script> tag** provide a way to get around the Same Origin Policy

```
<script src=http://otherdomain.com/data.js>
```

```
</script>
```

- The **src** attribute of a script tag can be set to a **URL from any server**, and every browser will go and retrieve it, and read it into your page
- So, a script tag can be set to point at a URL on another server with JSON data in it, and that JSON will become a global variable in the webpage
- So JSON can be used to grab data from other servers, without the use of a server-side proxy
- Available in **HTML** since 1994

# JSON and Dynamic Script Tag “Hack”

- Using JSON, it is possible to get around the limitation that data can only come from a single domain (bypasses cross-domain)
- To do this one needs to
  - to find a website that returns JSON data, and
  - A JavaScript program that contains a JSONScriptRequest class that creates a **dynamic <script> tag** and its contents
- The implementation of this class can be found at the class website:  
[http://csci571.com/examples/js/jsr\\_class.js](http://csci571.com/examples/js/jsr_class.js)
- The most important line in the script (the "hack") is the following one:  

```
this.scriptObj.setAttribute("src", this.fullUrl + this.noCacheIE);
```

which sets the src attribute of the <script> tag to a new URL

## Source Code for jsr\_class.js

```
// Constructor -- pass a REST request URL to the constructor
function JSONscriptRequest(fullUrl) {
    // REST request path
    this.fullUrl = fullUrl;
    // Keep IE from caching requests
    this.noCacheIE = '&noCacheIE=' + (new Date()).getTime();
    // Get the DOM location to put the script tag
    this.headLoc = document.getElementsByTagName("head").item(0);
    // Generate a unique script tag id
    this.scriptId = 'JscriptId' + JSONscriptRequest.scriptCounter++; }

// Static script ID counter
JSONscriptRequest.scriptCounter = 1;

// buildScriptTag method
JSONscriptRequest.prototype.buildScriptTag = function () {
    // Create the script tag
    this.scriptObj = document.createElement("script");
    // Add script object attributes
    this.scriptObj.setAttribute("type", "text/javascript");
    this.scriptObj.setAttribute("charset", "utf-8");
    this.scriptObj.setAttribute("src", this.fullUrl + this.noCacheIE);
    this.scriptObj.setAttribute("id", this.scriptId); }

// removeScriptTag method
JSONscriptRequest.prototype.removeScriptTag = function () {
    // Destroy the script tag
    this.headLoc.removeChild(this.scriptObj); }

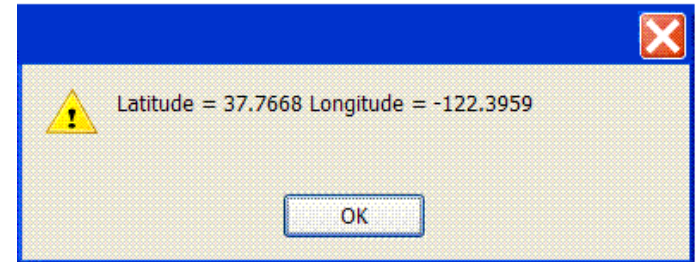
// addScriptTag method
JSONscriptRequest.prototype.addScriptTag = function () {
    // Create the script tag
    this.headLoc.appendChild(this.scriptObj); }
```

Critical line



# Example Using JSON and JSONscriptRequest class

```
<html><body>
// Include the JSONscriptRequest class
<script type="text/javascript" src="jsr_class.js"> </script>
<script type="text/javascript">
// Define the callback function
function getGeo(jsonData) {
    alert('Latitude = ' + jsonData.ResultSet.Result[0].Latitude + ' Longitude = ' +
        jsonData.ResultSet.Result[0].Longitude);
    bObj.removeScriptTag();
}
// The web service call
var req = 'http://api.local.yahoo.com/MapsService/V1/geocode?appid=YahooDemo&output=json&callback=getGeo&location=94107';
// Create a new request object
bObj = new JSONscriptRequest(req);
// Build the dynamic script tag
bObj.buildScriptTag();
// Add the script tag to the page
bObj.addScriptTag();
</script></body></html>
```



output

**buildScriptTag** creates

`<script src="getGeo({'ResultSet':{'Result':[{'precision':'zip',...>`

Adding the `<script>` tag to the page causes `getGeo` to be called  
And the JSON-encoded data to be passed to the `getGeo` function;  
The JavaScript interpreter automatically turns JSON into a  
JavaScript object, and the returned data can be referenced  
Immediately. See:

<https://www.xml.com/pub/a/2005/12/21/json-dynamic-script-tag.html>

# Tiingo.com Example

- Tiingo provides Stock Lookup and Stock Quote APIs. Results are in JSON format.
- To access the service, you do not need an Application ID.
- Stock End-of-Day Quote JSON REST call looks like this:

```
https://api.tiingo.com/tiingo/daily/aapl/prices?startDate=2019-01-02&token=<token>
```

- **Response:**

```
[ { "date":"2019-01-02T00:00:00.000Z", "close":157.92, "high":158.85, "low":154.23, "open":154.89, "volume":37039737, "adjClose":157.92, "adjHigh":158.85, "adjLow":154.23, "adjOpen":154.89, "adjVolume":37039737, "divCash":0.0, "splitFactor":1.0 }, ... ]
```

- **Crypto Top-of-book JSON REST call looks like this:**

```
https://api.tiingo.com/tiingo/crypto/top?tickers=curebtc&token=<token>
```

- **Response:**

```
[ { "ticker":"curebtc", "baseCurrency":"cure", "quoteCurrency":"btc", "topOfBookData":[ { "askSize":21.55601545, "bidSize":726.29848588, "lastSaleTimestamp":"2019-01-30T00:19:34.777000+00:00", "lastPrice":1.894e-05, "askPrice":1.9e-05, "quoteTimestamp":"2019-01-30T00:44:34.209957+00:00", "bidExchange":"BITTREX", "lastSizeNotional":0.0010885247347072, "lastExchange":"BITTREX", "askExchange":"BITTREX", "bidPrice":1.894e-05, "lastSize":57.47226688 } ] } ]
```

# XMLHttpRequest Compared to the Dynamic Script Tag

	XmlHttpRequest	Dynamic script Tag	
Cross-browser compatible?	No	Yes	Dynamic script tag is used by internet advertisers who use it to pull their ads into a web page
Cross-domain browser security enforced?	Yes (*)	No	
Can receive HTTP status codes?	Yes	No (fails on any HTTP status other than 200)	The script tag's main advantages are that it is not bound by the web browser's cross-domain security restrictions and that it runs identically on more web browsers than XMLHttpRequest.
Supports HTTP GET and POST?	Yes	No (GET only)	
Can send/receive HTTP headers?	Yes	No	
Can receive XML?	Yes	Yes (but only embedded in a JavaScript statement)	If your web service happens to offer JSON output and a callback function, you can easily access web services from within your JavaScript applications without having to parse the returned data
Can receive JSON?	Yes	Yes (but only embedded in a JavaScript statement)	
Offers synchronous and asynchronous calls?	Yes	No (asynchronous only)	

\* CORS-compatible browsers allow cross-domain XMLHttpRequest



# Arguments against JSON

- JSON doesn't have namespaces
- JSON has no validator
  - Every application is responsible for validating its inputs
- JSON is not extensible
  - But it does not need to be
- JSON is not XML
  - But a JavaScript compiler is a JSON decoder

# Features that make JSON well-suited for data transfer

- It is both a human and machine-readable format;
- It has support for unicode, allowing almost any information in any human language to be communicated
- The format is self-documenting in that it describes structure and field names as well as specific values
- The strict syntax and parsing requirements allow the parsing algorithms to remain simple, efficient, and consistent
- JSON has the ability to represent the most general of computer science data structures: records, lists and trees

# eval() and JSON.parse() Security

- The eval() function is very fast. However, it can compile and execute any JavaScript program, so there can be security issues
- In general
  - your browser should not trust machines not under your absolute control
  - Your server must validate everything the client tells it
- To help guard the browser from insecure JSON input, use **JSON.parse()** instead of eval ; e.g., JSON.parse() is used this way  

```
var myObject = JSON.parse(JSONtext [, reviver]);
```
- The optional reviver parameter is a function that will be called for every key and value at every level of the result. Each value will be replaced by the result of the reviver function. This can be used to reform generic objects into instances of pseudoclasses, or to transform date strings into Date objects.

















# More on JSON.parse()

- JSON.parse() is included in ECMAScript since 5<sup>th</sup> Ed. and all recent desktop browsers (Chrome, Firefox 3.5+, IE 8+, Opera 10.5+, Safari 4+) and all mobile browsers. JSON.parse() compiles faster than eval(). See:

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/JSON/parse](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse)

## Browser compatibility

[Update compatibility data on GitHub](#)

													
	 Chrome	 Edge	 Firefox	 Internet Explorer	 Opera	 Safari	 Android webview	 Chrome for Android	 Firefox for Android	 Opera for Android	 Safari on iOS	 Samsung Internet	 Node.js
parse	3	12	3.5	8	10.5	4	≤37	18	4	11	Yes	1.0	Yes

What are we missing?



Full support

# More on JSON.parse (cont'd)

- In JavaScript there is a function called `JSON.parse`. It uses a single call to `eval` to do the conversion, guarded by a single regexp test to assure that the input is safe.
- The input object is traversed recursively, and various functions are called for each member of the object in post-order (i.e. every object is reviewed after all its members have been reviewed).
- For each member, the following occurs:
  - If reviewer returns a valid value, the member value is replaced with the value returned by reviewer.
  - If reviewer returns what it received, the structure is not modified.
  - If reviewer returns null or undefined, the object member is deleted.
- The reviewer argument is often used to transform JSON representation of ISO date strings into UTC format Date objects.
- Here is the original source code for `JSON.parse`

```
JSON.parse = function (text) {  
  return  
  (/^(\\s|[, :{}\\[\\]]|"(\\\\"|\\bfnrtu|[^\\x00-\\x1f"\\]) *"|-  
  ?\\d+(\\.\\d*)?([eE][+-]?\\d+)?|true|false|null)+$/).test(text)  
  && eval('(' + text + ')'); };
```
- According to Doug Crockford, “It is ugly, but it is really efficient.”
- For the actual source implementation and explanation of the behavior of the "reference implementation" of `JSON.parse` see
  - <https://github.com/douglascrockford/JSON-js>
  - (for applications that need to run on obsolete browsers)
- JQuery provides implicit `ParseJSON()` method and `responseJSON` property. See
  - <http://api.jquery.com/jquery.parsejson/> and <http://api.jquery.com/jquery.ajax/>

# Douglas Crockford Discusses AJAX

- Go to
- <https://www.youtube.com/watch?v=-C-JoyNuQJs>
- Start video at -27 minutes and run it as long as it is interesting
- He gets to JSON at -13 minutes

# JSONP

- JSONP or "*JSON with padding*" is a JSON extension wherein the name of a callback function is specified as an input argument of the call itself.
- It is now used by many Web 2.0 applications such as Dojo Toolkit Applications or Google Toolkit Applications.
- Further extensions of this protocol have been proposed
- Because JSONP makes use of script tags, calls are essentially open to the world. For that reason, JSONP may be inappropriate to carry sensitive data
- JSONP is supported by jQuery. See:

<https://learn.jquery.com/ajax/working-with-jsonp/>

# JSONP Example

- Consider the following <script> tag which includes a src attribute referring to a Google spreadsheet

<script

src="<http://spreadsheets.google.com/feeds/list/o03712292828507838454.2635427448373779250/od6/public/basic?alt=json-in-script&callback=listTasks>"> </script>

- the URL in the above script will return the following JSON result:

```
listTasks ({ "version": "1.0", "encoding": "UTF-8", "feed": {  
  "entry": [{  
    { "title":  
      { "type": "text", "$t": "Make google gadget w/spreadsheet example"},  
    "content": { "type": "text", "$t": "Status: Done" } },  
    { "title":  
      { "type": "text", "$t": "Do final project for class"},  
    "content": { "type": "text", "$t": "Status: NotStarted" }  
  } ] }  
});
```

- notice the src attribute's URL has alt=json-in-script and callback=listTasks. This tells google that it wants not just JSON (data) but JSONP (data passed as a parameter of the listed function listTasks).
- The about output is JSON data, surrounded by the Procedure call "listTasks", with the data as a "parameter".
- Try this in a browser:  
<http://spreadsheets.google.com/feeds/list/o03712292828507838454.2635427448373779250/od6/public/basic?alt=json-in-script&callback=listTasks>



## What is Actually Returned Recently (since 2010)

```
listTasks ({ "version": "1.0", "encoding": "UTF-8", "feed": { "xmlns": "http://www.w3.org/2005/Atom",
"xmlns$openSearch": "http://a9.com/-/spec/opensearchrss/1.0/",
"xmlns$gsx": "http://schemas.google.com/spreadsheets/2006/extended",
"id": { "$t": "https://spreadsheets.google.com/feeds/list/o03712292828507838454.2635427448373779250/od6/public/basic"},
"updated": { "$t": "2006-12-05T10:35:42.800Z"}, "category": { {"scheme": "http://schemas.google.com/spreadsheets/2006",
"term": "http://schemas.google.com/spreadsheets/2006#list"}}, "title": { "type": "text", "$t": "Sheet1"},
"link": { {"rel": "alternate", "type": "text/html",
"href": "https://spreadsheets.google.com/pub?key\u003do03712292828507838454.2635427448373779250"},
{"rel": "http://schemas.google.com/g/2005#feed", "type": "application/atom+xml",
"href": "https://spreadsheets.google.com/feeds/list/o03712292828507838454.2635427448373779250/od6/public/basic"},
{"rel": "self", "type": "application/atom+xml",
"href": "https://spreadsheets.google.com/feeds/list/o03712292828507838454.2635427448373779250/od6/public/basic?alt\u003djson-in-script"}},
"author": { {"name": { "$t": "pamela.fox"}, "email": { "$t": "pamela.fox@gmail.com"}}}, "openSearch$totalResults": { "$t": "2"},
"openSearch$startIndex": { "$t": "1"},
"entry": { {"id": { "$t": "https://spreadsheets.google.com/feeds/list/o03712292828507838454.2635427448373779250/od6/public/basic/cokwr"},
"updated": { "$t": "2006-12-05T10:35:42.800Z"}, "category": { {"scheme": "http://schemas.google.com/spreadsheets/2006",
"term": "http://schemas.google.com/spreadsheets/2006#list"}}, "title": { "type": "text",
"$t": "Make google gadget w/spreadsheet example"}, "content": { "type": "text", "$t": "status: Done"},
"link": { {"rel": "self", "type": "application/atom+xml",
"href": "https://spreadsheets.google.com/feeds/list/o03712292828507838454.2635427448373779250/od6/public/basic/cokwr"}},
{"id": { "$t": "https://spreadsheets.google.com/feeds/list/o03712292828507838454.2635427448373779250/od6/public/basic/cpzh4"},
"updated": { "$t": "2006-12-05T10:35:42.800Z"}, "category": { {"scheme": "http://schemas.google.com/spreadsheets/2006",
"term": "http://schemas.google.com/spreadsheets/2006#list"}}, "title": { "type": "text", "$t": "Do final project for class"},
"content": { "type": "text", "$t": "status: NotStarted"}, "link": { {"rel": "self", "type": "application/atom+xml",
"href": "https://spreadsheets.google.com/feeds/list/o03712292828507838454.2635427448373779250/od6/public/basic/cpzh4"} } } } } }
```

# JSONP Example, Cont' d

- When dynamically executed (e.g., using JavaScript's "eval" on it), it invokes the function listTasks :

```
function listTasks(root) {  
  var feed = root.feed;  
  var entries = feed.entry || [];  
  var html = [];  
  html.push('<ul>');  
  for (var i = 0; i < feed.entry.length; ++i) {  
    var entry = feed.entry[i];  
    var title = entry.title.$t;  
    var content = entry.content.$t;  
    html.push('<li>', title, ' (' , content, ') </li>');  
  }  
  html.push('</ul>');  
  document.getElementById("agenda").innerHTML =  
  html.join("");  
}
```

For more discussion of JSONP see  
<http://www.west-wind.com/Weblog/posts/107136.aspx>

- The "output" of such a function in the example would be like this:

```
<ul>  
  <li> Make google gadget w/spreadsheet example (Status: Done) </li>  
  <li> Do final project for class (Status: NotStarted) </li>  
</ul>
```

- and it is stored in the "html" variable.
- You then assign such HTML to the "innerHTML" of the element with Id "agenda", which then shows it on the web page.

# Python Includes JSON functionality

- Python includes an encoder and a decoder. JSON functions include
  - `json.dump` – serialize object as JSON formatted string
  - *class* `json.JSONDecoder` - simple JSON decoder
  - *class* `json.JSONEncoder` – extensible JSON decoder for Python data structures
- See: <https://docs.python.org/3/library/json.html>

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

# Compact Encoding JSON in Python

```
>>> import json
```

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

# Decoding JSON in Python

```
>>> import json
```

```
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
```

```
['foo', {'bar': ['baz', None, 1.0, 2]}]
```

```
>>> json.loads('"\\"foo\\bar"')
```

```
"foo\x08ar"
```

```
>>> from io import StringIO
```

```
>>> io = StringIO('["streaming API"]')
```

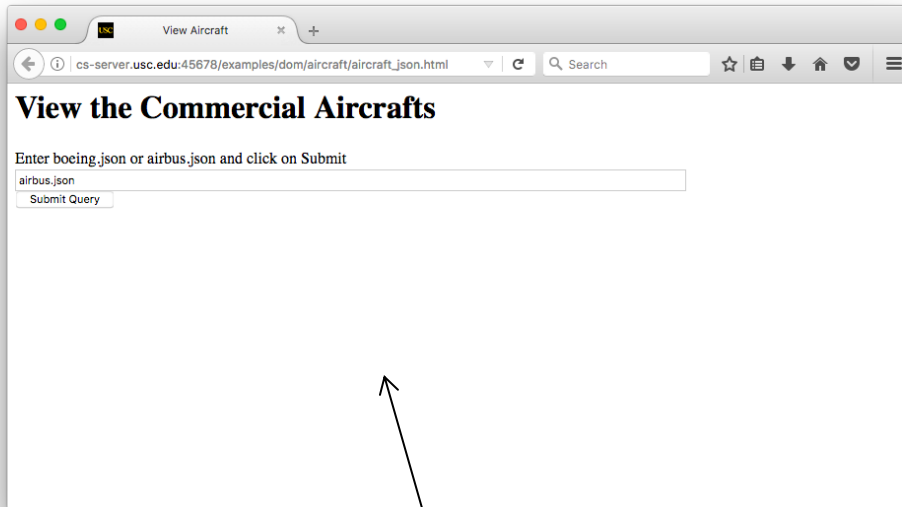
```
>>> json.load(io) ['streaming API']
```

```
['streaming API']
```





# JQuery JSON Support

- JQuery Ajax APIs provide extensive support for JSON and JSONP
- `jQuery.ajax([settings])` includes the following JSON-related settings:
  - **crossDomain** (default: **false** for same-domain requests, **true** for cross-domain requests) - If you wish to force a crossDomain request (such as JSONP) on the same domain, set the value of crossDomain to true. This allows, for example, server-side redirection to another domain.
  - **dataType** (xml, json, script, or html) - The type of data that you're expecting back from the server.
    - "json": Evaluates the response as JSON and returns a JavaScript object. The JSON data is parsed in a strict manner.
    - If json dataType is specified, the response is parsed using `jQuery.parseJSON` before being passed, as an object, to the success handler. The parsed JSON object is made available through the `responseJSON` property.
    - "jsonp": Loads in a JSON block using JSONP. Adds an extra `"?callback=?"` to the end of your URL to specify the callback.
    - If jsonp is specified, `$.ajax()` will automatically append a query string parameter of (by default) `callback=?` to the URL.
  - **jsonp** - Override the callback function name in a JSONP request. This value will be used instead of 'callback' in the 'callback=?' part of the query string in the url.
  - See <http://api.jquery.com/jquery.ajax/>

## Example 14: A Longer DOM Example in JSON



A screenshot of a web browser window titled "XML Parse Result". The address bar shows "about:blank". The page displays a heading "Airbus Aircraft Families" above a table. The table has 8 columns: Family, Aircraft, Seats, Range, Wing Span, Length, Height, and Image. It contains 4 rows of data, each with an image of the corresponding aircraft.

Family	Aircraft	Seats	Range	Wing Span	Length	Height	Image
A380	A380	555	15000km	79.80m	73.00m	24.10m	
A330/A340	A340-600	380	14600km	63.45m	75.30m	17.30m	
A300/A310	A300-600	266	7500km	44.84m	54.10m	16.54m	
A320	A321	185	5600km	34.09m	44.51m	11.76m	

Given a URL of a JSON file that describes a set of aircraft, re-format the data into an HTML page.

See: [https://csci571.com/examples/dom/aircraft/aircraft\\_json.html](https://csci571.com/examples/dom/aircraft/aircraft_json.html)

# airbus.json

```
{
  "catalog": {
    "title": "Airbus Aircraft Families",
    "aircraft": [
      {
        "Airbus": "A380",
        "Aircraft": "A380 ",
        "seats": "555",
        "Range": "15000km ",
        "Wingspan": "79.80m",
        "Length": "73.00m",
        "Height": "24.10m",
        "Image": "http://cs-server.usc.edu:45678/examples/dom/aircraft/A380.jpg"
      },
      {
        "Airbus": "A330/A340",
        "Aircraft": "A340-600",
        "seats": "380",
        "Range": "14600km",
        "Wingspan": "63.45m",
        "Length": "75.30m",
        "Height": "17.30m",
        "Image": "http://cs-server.usc.edu:45678/examples/dom/aircraft/A340.jpg"
      },
      {
        "Airbus": "A300/A310",
        "Aircraft": "A300-600",
        "seats": "266",
        "Range": "7500km",
        "Wingspan": "44.84m",
        "Length": "54.10m",
        "Height": "16.54m",
        "Image": "http://cs-server.usc.edu:45678/examples/dom/aircraft/A300.jpg"
      },
      {
        "Airbus": "A320",
        "Aircraft": "A321",
        "seats": "185",
        "Range": "5600km",
        "Wingspan": "34.09m",
        "Length": "44.51m",
        "Height": "11.76m",
        "Image": "http://cs-server.usc.edu:45678/examples/dom/aircraft/A321.jpg"
      }
    ]
  }
}
```



## HTML Code for the Initial Input

<h1>View the Commercial Aircrafts </h1>

Enter JSON file

<form name="myform" method="POST" id="location">

<input type="text" name="URL" maxlength="255" size="100"  
value="airbus.json" />

<br />

<input type="button" name="submit" value="Submit Query"  
onClick="**viewJSON**(this.form)" />

</form>

## viewJSON Routine

```
function viewJSON(what) {  
  var URL = what.URL.value;  
  
  function loadJSON(url) {  
    xmlhttp=new XMLHttpRequest();  
    xmlhttp.open("GET",url,false); // "synchronous" (deprecated because it freezes the page while waiting for a response) *  
    xmlhttp.send();  
    jsonObj= JSON.parse(xmlhttp.responseText);  
    return jsonObj;  
  }  
  
  jsonObj = loadJSON(URL);  
  jsonObj.onload=generateHTML(jsonObj);  
  hWin = window.open("", "Assignment4", "height=800,width=600");  
  hWin.document.write(html_text);  
  hWin.document.close();  
}
```

- See:

[https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous\\_and\\_Asynchronous\\_Requests](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests)

## generateHTML Routine

```
function generateHTML(jsonObj) {  
    root=jsonObj.DocumentElement;  
    html_text="<html><head><title>JSON Parse Result</title></head><body>";  
    html_text+="<table border='2'>";  
    caption=jsonObj.catalog.title;  
    html_text+="<caption align='left'><h1>"+caption+"</h1></caption>";  
    planes=jsonObj.catalog.aircraft; // an array of planes  
    planeNodeList=planes[0];  
    html_text+="<tbody>";  
    html_text+="<tr>";  
    x=0; y=0;  
    // output the headers  
    var header_keys = Object.keys(planeNodeList);  
    for(i=0;i<header_keys.length;i++)    {  
        header=header_keys[i];  
        if(header=="Airbus") { header="Family"; x=120; y=55; }  
        if(header=="Boeing") { header="Family"; x=100; y=67; }  
        if(header=="seats")  header="Seats";  
        if(header=="Wingspan") header="Wing Span";  
        if(header=="height") header="Height";  
        html_text+="<th>"+header+"</th>";  
    }  
}
```

## generateHTML Routine (cont' d)

```
html_text+="/tr>";
// output out the values
for(i=0;i<planes.length;i++) //do for all planes (one per row)
{
    planeNodeList=planes[i]; //get properties of a plane (an object)
    html_text+="/tr>"; //start a new row of the output table
    var aircraft_keys = Object.keys(planeNodeList);
    for(j=0;j<aircraft_keys.length;j++)
    {
        prop = aircraft_keys[j];
        if(aircraft_keys[j]=="Image")
        { //handle images separately
            html_text+="/td><img src='"+ planeNodeList[prop] +"' width='"+x+"' height='"+y+"'></td>";
        } else {
            html_text+="/td>"+ planeNodeList[prop] +"/td>";
        }
    }
    html_text+="/tr>";
}
html_text+="/tbody>";
html_text+="/table>";
html_text+= "</bo> + <dy> </html>"; }
```